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
Abstract

Transactional memory (TM), the concurrency control mechanism is now accepted as mainline. It is present not only in GCC, one of the most important open-source C/C++ compilers today, but also in most popular commodity hardware. While innovation can still be expected to emerge, TM is already a standard feature in the programmer toolkit.

Yet, the naive programmer trying to use TM will be disappointed to find that its limitations render it inefficient for many common workloads. To address this inadequacy, this thesis proposes use of Consistency Oblivious Programming (COP) [1] to design data structures with TM, both in hardware and in compilers. This advance yields both good performance and scalability in programs where TM by itself is not 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 22
   4.1 STM in the GCC Compiler 23
   4.2 TM-Pure and Suspended Semantics 24
   4.3 COP Composing using Suspended Transactions 25
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 ....................... 17
3.3 RB-Tree insert operation Fit in generic COP template ............... 18
3.4 RB-Tree insert ROP in generic COP template ....................... 18
3.5 RB-Tree insert ROP verification in generic COP template ........... 19

4.1 STM COP Template ................................................ 26
4.2 Execution time of the Vacation and the Intruder benchmarks ........ 30
4.3 RB-Tree with variable number of threads .......................... 32
4.4 RB-Tree with 5 operations per tx and 8 threads ..................... 33
4.5 RB-Tree with 4 threads and 20% update rate ....................... 33

5.1 HTM COP template .............................................. 36
5.2 An example dynamic cache-oblivious B-tree. ........................ 42
5.3 Associativity limits on Haswell HTM .............................. 44
5.4 RB-Tree various transaction sizes ................................ 45
5.5 RB-Tree contentious workload ................................... 45
5.6 PMA performance for read-only operations ......................... 47
5.7 Cache Oblivious B-Tree performance ............................. 48
Chapter 1

Introduction

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

If data can be partitioned so that each process operates on its own piece of data, parallelism will be able to operate freely. If data cannot be easily partitioned or if there are concurrent operations being conducted on the same part of the data by different threads, then the 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 [7] 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

<table>
<thead>
<tr>
<th>Lock synchronization</th>
<th>TM synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>input : parameters</td>
<td>output: rc</td>
</tr>
<tr>
<td>output: rc</td>
<td>input : 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 maximum speedup.

There are two ways to make $P$ larger, i.e., decrease 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 operations such as these as one atomic block without reintroducing the global lock.

The second option is to keep the semantics of the single global lock, but to minimize the serialization involved by monitoring 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) [35]. 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 access involves high overhead, even when done in hardware, and to treat all operations as unstructured sets of memory access makes it hard to identify false connections, i.e., benign situations that occur when one operation writes an address that was read by another operation.

COP can integrate these two options. On one hand, COP uses TM to achieve SGLA composability 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 STM compiler 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 conflict, and eliminates the overhead associated with it.

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) [8], and a Leaplist [9], 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 unacceptable overhead in software or unacceptable 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 facilitates 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 STM support in the compiler, i.e., how the COP operation are executed using features available in the compiler and how these COP operations can be composed and integrated into an arbitrary STM transaction.

Chapter 5 describes the adjustment of COP to the 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 highlights the contribution of COP to concurrent programming.

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

Background

This chapter provides information on the two integrated lines of research in COP: concurrent data structures and TM and then explains how they are synergized by COP.

2.1 Terms in Concurrent Programming

This section explains concepts used throughout the thesis. Concepts, such as caches, NUMA, and specific types of locks while important to concurrent programming, are not relevant to this thesis and hence are not specifically mentioned; nonetheless they can be learned from [47].

Correctness. The notion of linearizability [41] refers to an operation (or set of operations) that appears to the rest of the system to be occurring instantaneously. Linearizability is a common correctness criterion that implies that an algorithm is operating according to its sequential specifications.

Progress. Concurrent algorithms can be identified according to the amount of progress that they guarantee. We say an algorithm is wait-free [44] if it ensures that every thread will proceed in spite of an arbitrary delay (or even a failure) in other threads. It is lock-free if it ensures that some thread will always proceed. If an algorithm guarantees progress only to a thread that eventually executes in isolation, it is obstruction-free. The lowest level of progress occurs when locking is involved. At this level, there is no guarantee of progress.
**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 carrying out the code and must *release* it upon exiting the protected code. While there are many different types of locks that are tailored for different hardware architectures and different application requirements, yet each entails reciprocal exclusion, i.e., only one thread at a time is allowed to acquire the lock and execute the code. Use of a single global lock is simple and allows arbitrary code to execute atomically. Splitting a single global lock into multiple locks that protect different objects or states requires effort on the part of programmers and is prone to error. Furthermore, objects either have to expose locks that are stored in the object, or callers have to manage the locks for components that they use; both are strategies that break information hiding. If operational 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 (see Section 2.2). These methods, while efficient are encapsulated in the algorithms and this 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 maintaining a lock or a shared counter. This problem has been solved by the introduction of the construct of a hardware transaction, wherein any compound statement can be atomically executed, but, most architectures and algorithms in the literature use CAS atomic instructions. CAS receives the parameters: address, expected value, and new value. Then, if the address contains the expected value, it sets the address to the new value and returns success, otherwise it returns failure. The reading and the writing of the address are thus made atomic by hardware, i.e., no concurrent instruction can overwrite the expected value if a CAS has 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, therefore it 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 at 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 for the creation of scalable concurrent data structures. Locking the minimum set of objects per operation permits many operations to 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 necessary to ensure both high performance and freedom from deadlocks.

A well-known lock-based technique for creating such operations is hand-over-hand locking which decreases the duration during which locks are held by releasing locks on nodes that no longer affect the correctness of the operation. Application of this technique to B-Trees is described in [12], but it can also be trivially applied to simpler concurrent DAGs, such as linked lists and binary trees. [33] explains how to use shape analysis to automate fine-grain locking algorithms.

2.2.2 Non-Blocking

When the number of threads in an application is greater than the number of processors, the algorithm must handle unpredictably long delays due to OS scheduling. If a thread is blocked while holding a lock on the head of a linked list or the root of a tree, for example, all the other threads cannot use the data structure. To handle these situations lock-free algorithms have been designed for traversal of nodes. Auxiliary nodes [59] or marks [51] are employed to allow verification of validity. Updating operations, such as
2.2. CONCURRENT DATA STRUCTURES

an insertion or deletion use one CAS operation to change the contents data structure. In this way, the data structure is always in a valid state and can be continuously accessed, regardless of the workload or the state of the working threads. The scope of lock-free data structures has evolved to include unbalanced binary trees [27], and k-ary trees [18] that have been adjusted to support lock-free range queries in [16]. Recently, a technique for lock-free balanced trees has been introduced in [17].

2.2.3 Lazy Synchronization

Lazy synchronization is a hybrid of lock-free and fine-grained locking. Lazy operations lock only a minimal set of nodes and only for updates. Like lock-free data structures, lazy implementations maintain the data structure in such a state that unsynchronized traversal will never crash for an uninitialized pointer, nor will it enter into an infinite loop. This feature allows a search operation to ignore all concurrent operations. Furthermore, the data-structure is designed such that when an unsynchronized search completes, it returns linearizable output, e.g., if key, k was found, then k was in the set at some time during the search. In the case of an update, only a minimal set of nodes in the data structure is locked before the algorithm writes new information. In the lazy-list [40] algorithm, traversal is executed with no locking. Only the location where an update finally occurs is locked. Lazy algorithms, like lock-free ones add a mark to a node in the data structure [40, 43], which is used to verify that that node is still part of the shared data structure.

Performance evaluation of [40] shows that by adding lazy searches to the lock-free list of [51], performance is improved by an order of magnitude. The basic technique of lazy linked lists described above was developed in order to design a high performance lazy Skiplist [43] which allows for fast searches. [14] presents a lazy, relaxed balance AVL tree. Although tree traversals are not locked, synchronization is performed by checking the timestamp on each node encountered. Before modifying a node, the algorithm locks it and increments its timestamp. A similar method was later used in [5] to create an efficient splay tree.

In COP the lazy technique is extended to even more data structures, such as a B-Tree and Leaplist, while at the same time avoiding the continuous verifications of [5, 14]. In addition, as TM is used, operations upon COP data structures are composable.
2.3 History of TM

This section only highlights some milestones and directions in TM research and their adoption by the industry. Relevant TM algorithms and API are given in detail in Chapter 4 and Chapter 5. A more comprehensive survey of TM can be found in [37]. TM research began in 1993 in [46] where TM was suggested as a hardware feature. In 1995, STM was proposed by [57]. These two papers initiated TM research. However in both these papers, the notion of a transactions was static, i.e., all locations accessed had to be declared at the beginning of the transaction.

In 2003, [45] and [38] offered the first dynamic STM which also represented the first step toward TM compiler support by adding TM constructs to Java. These realizations of STM place no constraint upon addresses accessed in a transaction. This has allowed it to be applied 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 [56]. 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, [28] showed that efficient STM may use locks in its implementation and [26] coupled locks with its version to create the TL2 algorithm.

In 2007, [11, 32, 53] integrated TM into the C and C++ programming languages, relying upon the compiler to translate the developer’s marks of a transaction, into its TM realization so that an application developer does not need to call the TM library directly. In [32] the first publicly-available TM support for a C/C++ compiler was introduced.

Although [46] pioneered the idea of HTM more than twenty years ago, it materialized only quite recently in Sun’s Rock CPU [20] (which was later canceled). It was also incorporated into Intel in 2013 [48] and later IBM [19]. As a result of the collaboration between several companies since 2008, transactional language constructs for C++ with their draft specifications have already been proposed for standardization. Since 2012, TM has been developed under the umbrella of Study Group 5 of the ISO C++ Committee. GCC supports most of this draft specification in its 2012 4.7 release [55].
2.4 Properties of TM

The TM paradigm provides the programmer with an abstraction: the transaction [34]. Transactions make concurrent programming as easy as using critical sections and potentially as efficient as fine-grained locking. Various realizations of TM attribute different safety and correctness characteristics to transactions, such as:

Strong isolation (SI) [50], in which transactions are isolated, both from other transactions and from concurrent non-transactional access. SI is the strongest correctness guarantee. SI implies the qualities of opacity and privatization that are discussed below. It is, however, impractical to force SI into STM [22], as that would require that every non-transactional access be converted into a small transaction, which in turn introduces too high an overhead. In HTM, SI is the standard guarantee because hardware monitors all access transactional and non-transactional and can efficiently react to conflicts among them. If a non-transactional write hits a location that is concurrently being used in a transaction, the transaction aborts.

Serializability [54], another safety and correctness characteristic means that all committed transactions in 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 one with no concurrency among transactions.) Serializability is a commonly required property in data-bases, as well as in TM, but with TM, internal states of transactions need to be restricted.

Opacity [36] is serializability with the additional restriction that operating, non-committed transactions be prevented from accessing inconsistent states. Capturing opacity in STM is not trivial and involves the overhead of maintaining versions and revalidations. Sometimes, opacity is compromised to improve performance [21].

The privatization property [58] 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 standard STM algorithms [26, 30] require that a writing transaction must execute a quiescence barrier [24] after the commit to ensure privatization.
CHAPTER 2. BACKGROUND

2.5 TM’s Inherent Limitations

The introduction of TM into compilers and hardware might seem to imply that transactions are easy to use: that a programmer need only mark the atomic sections with transaction delimiters. While the creation of efficient concurrent data structures might seem to be 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—in practice TM has fundamental shortcomings. The impression that the many techniques in concurrent data structures developed over the past thirty years can each be discarded and replaced by TM is false due to problems encountered especially in hardware and in the compiler.

A TM transaction maintains read and write sets, either through software (STM) or in hardware (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 [23, 26, 57]. This rigid order of operations imposes certain limitations, both in performance and in adaptations. As the TM must log every access, it must either fit in the HTM cache or be explicitly logged by software. If a transaction overflows the HTM cache, it fails. This logging forces STM to call a function with every access, i.e., instrumentation. Instrumentation forces memory access to consume many more resources than that demanded by the original load and store instructions.

Once a TM transaction has logged the accessed address, it continuously monitors all future accesses of that address in the system so as to verify that the address has not been externally modified. If the monitored address is written by another concurrent thread, the conflict registered by the monitoring operation causes the transaction to fail. Yet, 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

The practical problems encountered by TM have motivated research into TM algorithms and TM-friendly data structures.

The boosting [42] family of STM algorithms uses TM only as a composition method for operations. It assumes that every method has an inverse, and accordingly it creates transactions 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 need only deal with semantic conflicts while the actual interleaving of access is managed by underlying methods. This approach yields high performance, but is limited to reversible methods and does not benefit from hardware and compiler support.

Another STM algorithm that only supports composition is transactional predication [15]. Like [42], it relies upon existing concurrent libraries, but instead of logging reverse operations, it logs locations where data is to be updated by a transaction. If a transaction fails, the value contained in the memory location remains unchanged or in the case of an insertion is replaced by an empty slot. This approach yields highly complicated algorithms and as no method has yet been offered to release empty slots, it wastes an uncontrollable amount of memory.

A set of algorithms has been developed to reduce the overhead of STM by relaxing some of the correctness requirements of the original TM. In this manner, some overhead and false conflicts are prevented. Elastic [31] and view transactions [6] do not log some of the accessed addresses, thus eliminating part of the transactional workload. While these algorithms do indeed improve performance, they still preserve much overhead and grant an application access to transaction logs. This not only places a burden upon the developer, but is also incompatible with GCC compiler architecture. These algorithms therefore are unlikely to be part of any practical STM solution.

In [25], small HTM transactions are used to create concurrent algorithms for queues that are simpler than their non-transactional counterparts. While this method demonstrates the power of HTM, it is not generally used.

TM-friendly data structures embody other conflict reduction techniques. [20] decouples the balancing of a binary tree from updating operations which avoids some of the conflict in highly contentious workloads. These techniques do not, however mitigate the high overhead associated with TM.

To benefit from compiler support for STM and from HTM, COP [4] and later, [61] and [60] leave the read-only prefix of the atomic operation out of the transaction. This lazy approach extends the usability of TM, while at the same time utilizing hardware and compiler support.
2.7 COP in a Nutshell

COP takes advantage of both TM and contemporary developments in data structures. TM generalizes the lazy approach (see Section 2.2.) permitting a relatively simple conversion of sequential operations to efficient scalable, composable concurrent operations. COP enables developers to design operations for complex data structures that do not yet have any known concurrent version by using the data structure algorithm to extract the read-only prefix (ROP) of the operation and verify that this prefix will not crash or hit an infinite loop when no synchronization is involved. This prefix returns an output that is either the output of the operation or input necessary for the completion of the operation. Completion, here, means any updating required by the operation. In an insert function for an RB-Tree, for example, the update may include the connection of a new node to the tree and balancing it. After extracting the ROP, the developer uses a TM transaction to perform two actions atomically: verifying that the ROP output is valid and completing the update. From 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 thereby eliminating the overhead of the transaction. This implies that the ROP will perform un-instrumented access to shared memory in STM and that this access does not leave a transaction footprint in HTM and is not subsequently 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 executed and generated output, a transaction starts or continues, verifying the output and using it to perform any updates.

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

The results in this chapter have been published in an article in OPODIS’11 [4] co-authored with Afek and Shavit.

3.1 The COP Template

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

General COP Template for Function $\kappa$

7 \textbf{start\_transaction} 3;
8 ANY CODE;
9 \textbf{suspend\_transaction}:
10 $\kappa$ROPOutput $\leftarrow$ $\kappa$ROP();
11 \textbf{resume\_transaction}:
12 if $\neg$($\kappa$Verify($\kappa$ROPOutput)) then
13 \hspace{1em} \textbf{abort\_transaction}:
14 $\kappa$Complete($\kappa$ROPOutput);
15 ANY CODE;
16 \textbf{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.

The TM transaction is delimited by \textbf{start\_transaction} and \textbf{end\_transaction} keywords, however, inside the transaction, there can be non transactional blocks, which are delimited by \textbf{suspend\_transaction} and \textbf{resume\_transaction} keywords. In Chapter 4 we manage to add suspended code segments to STM compiler support. In [3] we discuss adding non transactional loads, i.e. allowing suspended mode for ROP, to existing hardware.

If the non-transactional code generates output which is input to the transactional code following it, this output must be consistent, otherwise the transaction must ignore this input and execute the non-transactional code again, this time in transactional mode. The output might be consistent if it were generated on the same snapshot that the transaction is working on. Verifying consistency of output is done after resuming the transaction and must not introduce too much overhead in order to maintain the advantage of using COP. Rerunning non-transactional code in transactional mode is not a good verification method. Efficiency of verification can be derived from knowledge of the application. For example, if we know that the application never splits a tree and executes only insertions and deletions, then checking that a node is still connected to
its parent is sufficient verification for its existence in the tree.

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 synchronization, even though it resides in a transaction. Thus $\kappa$ROPOutput might be inconsistent or wrong due to conflicts with a concurrent transactions.

After calculating $\kappa$ROPOutput, we resume the transaction (Line 11), and call $\kappa$Verify($\kappa$ROPOutput) (Line 13). If $\kappa$Verify sees that $\kappa$ROPOutput is inconsistent, it aborts and retries the transaction. If $\kappa$Output is consistent, the transaction executes $\kappa$Complete($\kappa$ROPOutput). $\kappa$Complete($\kappa$ROPOutput) uses $\kappa$ROPOutput in order to perform any updates under the presumption that $\kappa$ROPOutput is correct.

If the transaction aborts due to an explicit abort_transaction or because of a conflict, it automatically retries it and if there are too many retries, the TM mechanism executes it on its own to verify progress just as if it were another transaction without any COP operations.

3.1.2 Correctness Proof Method

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

**Property 1. Transactional Regular Registers:** transactional locations are regular-registers \cite{47}, 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 in transactional regular registers.

Transactional regular registers are mean that the ROP cannot 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 ensure that: It completes without fault regardless of concurrent executions and that it finishes within a finite number of steps if it executes alone.

Obliviousness is progress related, if $\kappa$ROP() crashes or gets stuck in an infinite loop, no work is done. The following two properties ensure the correctness of the COP
CHAPTER 3. THE COP METHODOLOGY

operation.

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

**Property 4. Separation:** $\kappa$Complete uses $\kappa$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, $\kappa$ROPOutput remain internally consistent until the commit operation, 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 executing in a transaction that is semantically protected with all barriers necessary inserted by the TM.

Now, in implementing a COP version of a function $\phi$, we need only show $\phi$ROP, $\phi$Verify and $\phi$Complete. If, for example, we want to demonstrate a COP implementation of an RB-Tree Insert function, we present ROP, InsertVerify and InsertComplete. After creating the COP version, we show that it has the three correctness properties described above.

### 3.2 A COP Based RB-Tree

The canonical example of a COP data structure is the RB-Tree with chained leaves as introduced in [4]. After fitting the algorithm into the generic COP template, it is adjusted to the GCC compiler in Chapter 4 and adapted to the Haswell HTM block in Chapter 5.

In Figure 3.2, there are two concurrent search operations that start a search for an arbitrary key, Key 26 in an unbalanced RB-Tree. One is a COP operation which does a read-only prefix operation in a non-transactional context and the other is a plain TM operation which is in transactional mode. When both searches reach Key 27 the
3.2. A COP BASED RB-TREE

Figure 3.2: COP and TM search for key 26 in an RB-Tree

tree is balanced and Key 27 becomes the root of the tree. The COP search, which is not in a transactional context continues and reaches the leaf that holds Key 26. By contrast, the plain TM search, which is in a transactional context from the start fails right after balancing. This is because the search traverses the right pointer of Key 20 at the beginning of the search and in balancing it modifies that pointer. In addition, the balancing changes the color of Key 20 from black to red. As both color and pointer are located on the same node and thus probably in the same cache line, changing the color is, by itself, enough to cause the TM search to fail. After COP completes its non-transactional search, it resumes the transaction in order to verify that it got a valid result. Note that when the TM search failed, it lost the whole transaction, not just the "search for Key 26" operation. If, for example, Key 26 was a product of a heavy prior operation, that operation would be lost as well, while the transaction using the COP operation is able to continue.

3.2.1 Algorithm

The function chosen to be inserted into the COP template in Figure 3.3 is an insertion of a new value into an RB-Tree: Insert(RB-Tree T, Key K, Value V ). The Insert is split into three operations: InsertROP() which returns InsertROPOutput, InsertVerify(InsertROPOutput) which verifies the output of InsertROP(), and InsertComplete(InsertROPOutput) which performs the update.
RB-Tree insert operation

\begin{verbatim}
17 start_transaction;
18 ANY CODE;
19 suspend_transaction;
20 InsertROPOutput ← InsertROP(RB-Tree, Key);
21 resume_transaction;
22 if ¬(InsertVerify(InsertROPOutput)) then
23     abort_transaction;
24 InsertComplete(RB-Tree, InsertROPOutput, Key, Value);
25 ANY CODE;
26 end_transaction;
\end{verbatim}

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

RB-Tree insert ROP

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

Figure 3.4: RB-Tree insert ROP in generic COP template
3.2. A COP BASED RB-TREE

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 \( N \) cannot 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 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 [26]. This only underlines the fact that exactly the same COP algorithm can also be used for Delete and Lookup functions.

3.2.2 Correctness

After placing the RB-Tree search algorithm into the COP template, it is only left to show that it has all the properties of COP correctness which signifies that it is serializable. It is assumed there is safe-memory reclamation which ensures that a node is not recycled until all tasks that access it have terminated. This can be achieved by using methods from [39].

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

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

**Proof.** When InsertROP reaches node, N, it traverses either its right or left pointer. N is either in the tree or was in the tree and got removed. If the node has been removed, its pointers are set to null, so the ROP stops. If N is in the tree, then there is a finite path from N to the leaf. The leaf pointer will be null, and InsertROP will stop. According to Property 1, if a tree update is concurrent with the InsertROP, it will see either a null or a valid pointer.

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

Let N be the node returned from InsertROP. InsertVerify must determine if N is in the tree and if it holds the key searched for or if the key is not in the tree, but N holds either its potential successor or predecessor. As InsertVerify executes in the context of a transaction, it sees T updates as atomic operations. For example, if the node live mark is false, then it can already be determined that the node 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→right is null, K’s node can be connected to it as N right son. If N→right is not 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 **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 that has been verified. As the ROP did not write any global data, the only information it can pass to the complete function is the parameters.
3.2. A COP BASED RB-TREE

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

As we have now proven that all COP RB-Tree functions have the **Obliviousness**, **Verifiability** and **Separation** properties, we have shown the following:

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

STM with COP

Composing a number of transactions into a single atomic transaction is an important feature of transactional memory (TM) that is supported by many STM (Software Transactional Memory) implementations. This composition, however, typically results in long transactions with increased probability of contention among the various threads.

In consistency oblivious programming (COP), the read-only prefix of a data structure operation is performed outside the context of the TM transaction. The operation is then completed by a transaction that verifies the prefix output and performs updates. In STM, this strategy effectively reduces much overhead and potential contention.

In this work we emphasize the importance of transaction-suspension which enables performance of non-transactional memory access 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-TM and integrate COP into TM applications. We also support TM-Safe memory reclamation in transactions containing COP operations by adding privatization before a transaction-abort to the GCC-TM library.

Statistical counters added to GCC-TM show that using COP reduces the number of aborts and the number of transactional memory read operations.

The remainder of this chapter is organized as follows: Section 4.1 briefly describes use of GCC-TM intrinsics; Section 4.2 compares GCC-TM’s TM-Pure attribute and IBM’s Power8 transaction suspension operation; Section 4.4 describes the composition of transactions using COP and transaction suspension; Section 4.4 presents our evaluation. The results found in this chapter were published in an article PODC’13 [9] coauthored with Shavit and Suissa and in DISC’14 [10] coauthored with Suissa.
4.1 STM in the GCC Compiler

In GCC-TM, the transaction code is encompassed in a _transaction_atomic{} block. Inside this block, all accesses to shared memory are instrumented, i.e., instead of a plain load and store machine instructions, a GCC-TM library function is called with the corresponding address (and a value upon a write access). Depending on the GCC-TM mode, the function can either lock the location for writing, log it for future verification or rollback, or verify the timestamp version of the address.

GCC-TM supports various STM implementations, each having its own characteristics. An STM can use either a write-through or a write-back strategy. 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 commit (encounter-time locking), abort if the address is already locked, or log the address to be locked during commit-time.

Previous STM research [29, 30] showed that the most efficient and scalable STM is that based on multiple-locks and uses write-through and encounter-time-locking semantics. This is the default algorithm in the GCC-TM library. Note that a write-through implementation must use encounter-time locking to maintain isolation.

During compilation, the compiler infers which 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 _transaction_safe or _transaction_pure [55]. _transaction_safe functions are considered a part of the transaction, and thus a transaction that executes this function will issue the TM library functions when accessing memory locations. _transaction_pure functions, on the other hand, are never instrumented. In GCC-TM, the programmer
can decide to execute a code segment without instrumentation, although this decision can cause synchronization bugs.

A transaction can also call `__transaction_cancel`,

which aborts the transaction and undoes all its updates as though the transaction had never executed. The STM algorithm is designed for "best effort:" if, after a certain number of attempts, a transaction repeatedly aborts, it moves to serial execution mode in which it takes a global lock and pessimistically executes the code without concurrency.

4.2 TM-Pure and Suspended Semantics

When adding suspended mode to GCC-TM, we found that undocumented TM-Pure functions in write-through mode have similar semantics to those of the future POWER architecture of the HTM block [19] suspended state which is marked by two newly introduced instructions: `tsuspend` and `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 STM, it is already locked (`encountertime` locking) and the updated value is in the memory address (`write-through`). This means that a load instruction from an address that had been previously written by a transaction will return the value written during the transaction.

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

In case an STM transaction is aborted by another transaction during TM-Pure function execution, it detects the abort only upon returning from executing the TM-Pure code.

**Semantic 3.** Initiation of a new transaction is prevented.

Initiation of a new transaction in a TM-Pure code segment is allowed, although it can be prevented at compile time. We rely on the developer not to misuse this option.
The developer who is writing the shared data structure library should be sufficiently experienced to be aware of all kinds of concurrency bugs. That said, COP is still a relatively simple way to achieve efficient and composable modules.

### 4.3 COP Composing using Suspended Transactions

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 that the ROP execution cannot generate a segmentation fault, a floating exception, or even an infinite loop.

2. Start transaction T.

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

4. Complete the cop operation. In this stage we simply perform the update part of the transaction, as if it were not used in the COP methodology.

5. Commit the transaction T.

Verification in step 3 should introduce minimal additional overhead in order to preserve high performance. Rerunning the ROP in transaction mode, for example, might not be advisable.

In order to compose COP operations when transaction suspension is not available and avoiding instrumenting the ROP code block, the solution of executing all the ROP parts of composed operations before starting the transaction has been proposed and then inside the transaction verify results and complete updates, [60]. This, however, allows for composition only if an operation does not write data that may later be accessed by another operation in the same transaction.

To demonstrate this restriction, we divide each $o^k$ operation to $o^k_{ROP}$ and $o^k_{VC}$ (verify and complete). Assume $o^1$ precedes $o^2$, and $o^1$ is writing data that $o^2$ is reading. In [60], transaction $T$, which executes $o^1$ and then $o^2$, will execute in the following order:
CHAPTER 4. STM WITH COP

COP Template for Function $\kappa$

\begin{verbatim}
54 start_transaction;
55 ANY TRANSACTIONAL CODE;
56 tm_pure_start;
57 $\kappa$ROPOutput ← $\kappa$ROP();
58 tm_pure_end;
59 if ¬($\kappa$Verify($\kappa$ROPOutput)) then
60 $\kappa$ROPOutput ← $\kappa$ROP();
61 $\kappa$Complete($\kappa$ROPOutput);
62 ANY TRANSACTIONAL CODE;
63 end_transaction;
\end{verbatim}

Figure 4.1: STM COP Template

$\text{op}_1^{\text{ROP}} \rightarrow \text{op}_2^{\text{ROP}} \rightarrow T_{\text{START}} \rightarrow \text{op}_1^{\text{VC}} \rightarrow \text{op}_2^{\text{VC}} \rightarrow T_{\text{END}}$

As $\text{op}_2^{\text{ROP}}$ must execute before $\text{op}_1^{\text{VC}}$, $\text{op}_1$ will not see $\text{op}_2$ updates, so $T$ cannot be correct. It is not trivial to verify the independence of transactions, and grouping the operations parts together complicates the code.

Even if $\text{op}_1$ is not COP, but precedes $\text{op}_2$ in a transaction $T$, we must start $T$ before $\text{op}_1$, so $\text{op}_2^{\text{ROP}}$ will again be instrumented. If, for example, $T$ removes $V$ and then inserts $V$ to an RB-Tree with a COP operation, then this COP operation’s ROP will be instrumented, and this common scenario will not benefit from the usage of COP.

Using Transaction Suspension: In Figure 4.1 we show the template code block for creating a COP version of a serial operation, $\kappa$ using transaction suspension. As in any transaction, the code block starts by calling `start_transaction` (Line 54), and ends by committing using `end_transaction` (Line 63).

Computation of the read-only prefix (ROP) is done (Line 57) and its output is stored in $\kappa$ROPOutput. Encapsulating this code within `tm_suspend` and `tm_resume` instructions ensures execution without instrumentation.

If verification of the ROP output (Line 59) fails, the ROP code block is executed in transaction mode (Line 60) to ensure its completion. Finally, the COP operation is completed (Line 61) and the transaction is committed.

Another possible approach and one that complies with the original COP design is to explicitly abort the transaction (Line 60) in the case of failed validation. We note that
4.3. COP COMPOSING USING SUSPENDED TRANSACTIONS

in order to derive benefit from COP without transaction suspension, the transaction must be aborted and re-executed.

Executing ROP as a function with TM-Pure attributes and returning its output enables suspension of the transaction and avoids instrumentation.

Let \( \text{op}_{\text{ROP-PURE}} \) denote \( T \)'s execution of the ROP as a TM-Pure function. If \( T \) attempts to execute the COP operation \( \text{op}^2 \) after the COP operation \( \text{op}^1 \), it will go through the following steps:

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

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 \( T \), then \( \text{op}^2_{\text{ROP}} \), which executes after \( \text{op}^1_{\text{VC}} \) and performs its updates in the context of \( T \), can see these updates and \( T \) is correct.\(^1\)

4.3.1 Safety

To allow a ROP code block to complete without fault and generate verifiable output, we have to show that a load access from address A in a non-transactional ROP reads only those values that were previously written to A. This is true, because when the GCC-TM library unrolls a write-through transaction, it uses \texttt{builtin_memcpy()} which is optimized so that it copies the largest alignment possible. 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 in one atomic access. Thus, TM-Pure code operates on values that were previously written and not a conglomerate of old and new values.

4.3.2 Memory Reclamation

Two important TM-Safe [55] functions that can execute within transactions are \texttt{malloc} and \texttt{free}. These functions are made safe by privatization. If transaction \( T \) writes to memory, then before it returns, it waits for the termination of transactions that started before its commit [24]. As a side effect of privatization, in case \( T \) detaches a memory block from a data structure and then successfully commits, \( T \) can free that block. On the other hand, if \( T \) allocates a block of memory, \( M \) and then aborts, it

\(^1\)This follows from the semantics of the transaction suspension instructions, as described in Section 4.2.
CHAPTER 4. STM WITH COP

```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)

can free M without privatization, the reason being that the pointer to the tentative memory block is not exposed to other transactions. This however is violated where COP is involved: if the non-transactional ROP code block traverses a data structure and acquires a pointer to the newly allocated memory block and upon an abort of T and the freeing M, the ROP may still attempt to access unmapped memory. To prevent this from happening, we add privatization to the writing of transactions that are about to perform a rollback. If a transaction is read-only, it can free its tentative memory blocks unconditionally. If, however, the transaction updates a memory location, it has to perform privatization just as though it had committed. When comparing performance, the addition of this operation has negligible impact. With suspended mode and rollback privatization, malloc and free also become COP safe as memory is not recycled as long as a transaction is in progress (and COP operations are always encapsulated in transactions). One restriction upon this procedure is that memory allocation cannot take place in a ROP, because in case validation fails, the allocated node will not be freed since in this case the transaction is not aborted. However, as the ROP does not write to memory, it typically does not need to allocate or free it.

4.3.3 An RB-Tree Using COP and TM-Pure

We ported the RB-Tree from [4] into the COP-STM template ((Figure 4.1)). Listing 4.1, presents a C language lookup function. The function is the ROP coding for trans-
actional search, insert, and remove operations of an RB-Tree. The lookup function receives a pointer to RB-tree, s and a key to look for, k. If k is not found in the tree, the function returns a leaf that is either a potential successor to or predecessor of k, otherwise it returns the node that contains k. Note that this function is given the TM-Pure attribute ensuring that it executes without instrumentation. After a transactional operation (either a search, insert, or remove) receives a node from the TM-Pure lookup, it returns to transactional mode to verify that this node is valid, and if the outcome is positive, it complete the update to the data structure. The verification has transaction-safe attributes which ensure that all shared access in that section are instrumented.

4.4 Integration and Evaluation

Our evaluation uses 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.

To reveal the advantages of COP-based applications, we first present the performance indicators of our methodology on a subset of STAMP [52] benchmark applications and then present the performance indicators on a set of targeted micro-benchmarks.

4.4.1 COP and TM-Pure in STAMP Applications

The applications in the STAMP TM benchmark suite [52] use a library of transactional data structures which include a queue, heap, an RB-tree, a linked-list, and a hash table. As the queue and heap operations use a very short ROP, COP provides little run-time advantage. The linked list is used mostly as a hash-table bucket and through iterators the ROP of its operations is also very short. Thus, during the verification phase, COP implemented linked-list does not alter workload performance. To highlight the performance impact of COP, an application that relies heavily on an RB-Tree is needed.
TRANSACTIONAL LOADS

Aborts Rate 0.5% 3.0%

<table>
<thead>
<tr>
<th>Transactional Loads</th>
<th>GCC-COP $0.4 \times 10^9$</th>
<th>GCC-STM $2.4 \times 10^9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aborts Rate</td>
<td>0.5%</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

Table 4.1: STAMP Vacation Statistics

The Vacation application is the only candidate in STAMP that satisfies this criterion. Note that Yada and Intruder applications also use an RB-Tree, but not nearly as extensively as Vacation.

Table 4.1 shows the number of transactional loads and aborts of the Vacation benchmark. We count the number of transactional loads while using a single thread. The number of aborts is measured when all eight hardware threads execute, to get the highest number for the benchmark. We observe that plain STM is performing five times more transactional loads than does COP. The number of aborts in plain STM is six times greater than that of COP, but since the amount of aborts in plain STM is only 3% of the transactions, so the impact on performance is insignificant.

Figure 4.2a presents the execution time of the Vacation benchmark. In this benchmark HTM gives the lowest execution time when using a single thread. However, the execution time increases when the number of threads is 2 or more, due to contention (as described in Section 4.4.2), and is higher than gcc-cop. The HTM configuration we use in this experiment is the default in GCC-TM, i.e., two retries. We note more retries can improve the performance of HTM, but this is not the focus of this chapter.

Comparing the gcc-ml-wt and gcc-cop shows that using COP can greatly reduce the execution time. Even when only a single thread is used, COP is about 33% faster than the default STM. This emphasizes the fact that COP prevents much of the overhead caused by the default GCC-TM STM. As the contention grows, using COP still gives better execution time, but to a lesser extent.

Examining the execution time of the Intruder benchmark in Figure 4.2b roughly
4.4. INTEGRATION AND EVALUATION

shows the same results. In this case, however, using COP also reduces the overhead of the RB-Tree, although the benefit of using COP is minor, as the RB-Tree operations are less dominant in this workload.

4.4.2 STM with COP vs. Plain TM

To better understand the performance of COP, we test the COP RB-Tree as a stand-alone application. The RB-Tree benchmark has two transactions that modify its content: insert and remove. Both perform lookup in the ROP code section and one read-only search which also performs lookup. All HTM transactions are retried up to 10 times to eliminate spurious aborts. If, however, an HTM transaction hits a capacity abort, we assume that the transaction cannot be executed in HTM and will directly fall back to use of a global lock.

The graphs include the following lines:

1. s-op: Number of operations per second using gcc-ml-wt.
2. s-con: Number of aborts using gcc-ml-wt, as counted in GCC.
3. c-op: Number of operations per second using gcc-cop.
4. c-con: Number of aborts using gcc-cop, as counted in GCC.
5. h-op: Number of operations per second using gcc-htm.
6. h-con: Number of conflict aborts using gcc-htm.
7. h-cap: Number of capacity aborts using gcc-htm.

In the following tests each transaction performs a number of operations whose execution time is then measured. If an abort occurs due to conflict the number of operations in the transaction is counted as half, under the assumption that on average the conflict occurs in the middle of the transaction, and as there is no indication from the HTM where it actually happened. If an abort is caused by an HTM capacity shortage, the full number of operations in the transaction is counted because the transaction is eventually executed in serial mode and must succeed. This also means that the number of capacity aborts is bounded by the number of successful operations. Cases where the h-cap and the h-op lines coincide indicate that all HTM transactions abort due to capacity limits.
Table 4.2: RB-Tree transactional loads

<table>
<thead>
<tr>
<th>Tree Size</th>
<th>GCC-COP</th>
<th>GCC-STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (1K)</td>
<td>$50 \times 10^6$</td>
<td>$88 \times 10^6$</td>
</tr>
<tr>
<td>Large (1M)</td>
<td>$36 \times 10^6$</td>
<td>$120 \times 10^6$</td>
</tr>
</tbody>
</table>

In all our benchmarks, the performance (throughput) of `gcc-cop` is better than that of `gcc-ml-wt`, mainly due to lower instrumentation overhead, as seen in Table 4.2, and the higher abort rate of default STM. When comparing RB-trees with 10 K and 1M elements, it was found that in the smaller, 10K-element tree COP removes lesser instrumentation due to shorter ROP, but the lower abort rate of COP contributes to its better performance. In the larger, 1M-element tree while there is less contention, the ROP is longer so COP eliminates more instrumentation, which again makes it a better fit.

Figures 4.3a and 4.3b show the number of operations per second while varying the number of threads with an RB-Tree of 1K elements, 20% update rate and 10 operations per transaction, and 1M elements, 10% update rate and 5 operations per transaction, respectively.

When the tree is smaller (1K elements), default STM has a higher abort rate as compared to COP. Thus the throughput of COP is better for every increase in the number of threads. On the other hand, when the tree is large (1M elements), the small amount of instrumentation overhead also works to COP’s advantage. The same analysis holds in Figure 4.4a and Figure 4.4b which vary the update rates on the small and large trees. On the small tree (Figure 4.4a), as contention increases, COP performs better due to its ability to resolve conflicts, while on the large tree (Figure 4.4b), COP throughput is higher due to its reduction of instrumentation overhead. Figures 4.5a and 4.5b show the throughput and abort rate when modifying the amount of operations per transaction for both small and large trees. Hyper-threading, which occurs when...
there are more than four threads reduces the HTM capacity by half and triggers more HTM aborts which in turn increase the advantage of COP and STM. However, this is a known problem in Haswell HTM, so here we compare COP and HTM without hyperthreading.

For a small tree (Figure 4.5a), HTM performs better than COP for all transaction sizes checked. For a 1K-element tree there is no capacity problem. Yet, we see that as the number of operations per transaction grows, COP maintains scalability while the performance of HTM degrades, due to the higher number of conflict-aborts. On the larger tree (Figure 4.5b) the throughput of COP is better than that of HTM, due to the greater number of HTM capacity-aborts (even without hyperthreading) in transactions containing more than eight operations.

Overall, COP improves HTM and STM scaling in all parameters, i.e., when contention or length of transaction increases, COP performance improves as compared to that of all other TM implementations.
Chapter 5

HTM with COP

Consistency-oblivious programming mitigates, and, often solves, a major problem of the current 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 in an article in TRANSACT’14 [8] coauthored with Kuszmaul.
5.1 Improving HTM Scaling with COP

Our HTM version of COP is similar to the 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 helps to overcome two important drawbacks of HTM: the limited capacity for transactional access and the inability to release items from read and write sets. COP reduces the number of memory accesses in a transaction, and thus makes it more likely to adhere to the space limitations of hardware. Since the footprint of an HTM transaction must fit into the cache and caches typically provide limited associativity, programmers may be surprised to find that transactions, even some with small footprints cannot be carried out. Our experiments show that by using COP, many operations can be composed into a single COP transaction without violating hardware resources, whereas the simple HTM version of the same transaction 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.

5.2 HTM COP Template

In this chapter, we use COP with the Intel Haswell RTM, and the intrinsics \texttt{xbegin}, \texttt{xend} and \texttt{xabort}, that were introduced in GCC-4.8. The \texttt{xend} commits a transaction while \texttt{xabort} terminates it with an abort. The \texttt{xbegin} returns an error code. Relevant error codes in the context of COP are found 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 \( \kappa \text{ROP}() \) (line 65). \( \kappa \text{ROP}() \) calculates \( \kappa \text{ROPOutput} \), in an unsafe mode, i.e., without any synchronization. Thus \( \kappa \text{ROPOutput} \) might be inconsistent and wrong, due to conflicts.
with concurrent operations.

After calculating \( \kappa \text{ROPOutput} \), we start a transaction in line 66, and call \( \kappa \text{Verify}(\kappa \text{ROPOutput}) \) in line 68. \( \kappa \text{Verify} \) will call \_\text{abort} if \( \kappa \text{ROPOutput} \) is inconsistent. If \( \kappa \text{ROPOutput} \) is consistent, we will continue the transaction to execute \( \kappa \text{Complete}(\kappa \text{ROPOutput}) \). \( \kappa \text{Complete}(\kappa \text{ROPOutput}) \) will use \( \kappa \text{ROPOutput} \) and perform any updates, considering that \( \kappa \text{ROPOutput} \) 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 [21].

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., \( \kappa \text{Verify} \) called \_\text{abort}, we must rerun \( \kappa \text{ROP} \) to get a correct \( \kappa \text{ROPOutput} \), otherwise, the abort must have been due to a conflict, so \( \kappa \text{ROPOutput} \) may well be correct, and the transaction has a chance to commit successfully, thus, we reuse \( \kappa \text{ROPOutput} \) 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 [4] to our COP template. Listing 5.1 shows the code for inserting into an RB-Tree, using C notation to make the exposition closer to the real code.

The algorithm for insertion, which was introduced in [4] 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))
      // Lock was held. Always try again.
      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;
    }
    // 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 [39] 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).
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 NULL. When the tree is empty, as, for example, at the first insertion, the verification will fail by following a 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 NULL, and then it never will happen again. Also note that it saves conditions in the rb_rop_verify, which is frequently called.

In the case that the transaction failed, there are four interesting kinds of failures handled in the else clause at line 13.
# define BAD_ROP 1

```c
inline void RBVerify(node_t * p, int K) {
    node_t *next;
    if (!p) _xabort(BAD_ROP);
    if (!p->live) _xabort(BAD_ROP);
    if (p->k != K) {
        if (p->k > K) {
            if (p->l != NULL) _xabort(BAD_ROP);
            if (p->prev->k >= K) _xabort(BAD_ROP);
        } else {
            if (p->r != NULL) _xabort(BAD_ROP);
            if (p->succ->k <= K) _xabort(BAD_ROP);
        }
    }
}
```

Listing 5.3: RB-Tree COP ROP Verify

1. The transaction could have failed because the lock was held. In this case, at line 14 we always retry the transaction, because, when the lock is released, we have every reason to hope that our transaction will succeed.

2. The transaction could have failed because the read-only-prefix gave a bad answer. In this case, at line 22 we retry a limited number of times, because we know there actually are conflicts occurring from other transactions in the tree.

3. The transaction could have failed in some other a way that gives us hope that retrying will help. It turns out that almost all conflict failures have a chance of succeeding on retry. At line 24 we retry if the status has the retry bit set, if it has the capacity bit set, or if the status is zero indication some other failure (such as a time slice interrupt occurred).

4. Finally either we have exhausted our retry budget or we believe that retrying won’t be helpful for some other reason, and at lines 27–30 we lock the tree, redo the prefix, complete the insertion without verification, and unlock the tree.

5.4 Cache-Oblivious B-Tree

We also tested a dynamic cache-oblivious B-tree (COBT) [13]. 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
5.4. CACHE-OBLIVIOUS B-TREE

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 [49].

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 3/4ths of the array contains values less than or equal to ‘U’.

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

The advantage of a COBT is that it can perform insertions and deletions in amortized time $O(\log_B N)$ without knowing the cache line size $B$. Thus this data structure is optimal and cache oblivious. Although the average cost is low, our implementation has a worst-case insertion cost of $O(n)$. It turns out that one can build a COBT in which the worse-case cost is also $O(\log_B N)$, but we 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.
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 4.4 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
5.5. **EVALUATION**

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
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
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.
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. When transforming theoretical design into practical application, TM has some built-in pitfalls that render it impractical when incompetently used, even in common workloads. Throughout this thesis, we show that the COP methodology enables TM to create concurrent objects that scale and maintain high performance, while simple TM versions of these same objects do not. A simple HTM COBT with 1M nodes could not execute even one search, while the 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 simple STM counterpart. In this thesis we construct a generic template to facilitate the creation of COP data structures and supply a simple way of checking that COP operations are correct.

Then we port the template to existing HTM and STM compiler support. One of the most appealing aspects of TM is software composition, namely, the ability to develop pieces of software independently and then compose them into applications that behave correctly in the face of concurrency. With COP, this requires a suspended mode which we define and examine in STM compiler support. We employ it to add COP data structures to a library used by the standard STAMP benchmark.
Chapter 7

Future Work

More research, both theoretical and practical is needed to develop new COP methodologies.

**TM infrastructure and COP.** To optimize COP algorithms, the compiler should provide developers with standard interfaces to statistics, such as aborts (as are already available in HTM) and transaction access counts. Preferably, the access count should be per definable segments of code. Once this information is publicly available, it will be necessary to construct a new measure of complexity for concurrent algorithms that factors in abort rates and type of access, i.e., transactional or non-transactional. To make COP operations composable in HTM, the hardware must support suspended mode. 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 too benefit from TM synchronization. There already are COP versions for different trees, linked-lists, skip-lists, and others, but the task remains to fit other graphs and useful structures into COP.

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

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


53


[18] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In OPODIS, pages 207–221, 2011. 2.2.2

[19] 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


[22] Luke Dalessandro and Michael L. Scott. Strong isolation is a weak idea, 2009. 2.4


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


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


[51] 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


[60] Lingxiang Xiang and Michael L. Scott. Composable partitioned transactions. In WTTM, 2013. 2.6, 4.3

הפקולטה למדעי מדעיים
יישה רימונד ובברלי סאקלר
בית הספר למדעי המחשב ע"י עביני בלצניצק

תכלות המ.endsWith מעﹺכתב יהוד

היבור ויהוה גלעד התחוד "דוקטור לפילוסופיה"
על די

הלל אבני

עגודה המקה זו נשעת בנוחיות

פרופ. יהודה אפק
פרופ. ניר שביט

הווש להנאות השל אוניברסיטת תל-אביב

 qed"
תמצית

כמו חומרי מורחב ליבת, מגוון בקרת מוקבילית המכונה עסקאות (COP) וזכרô ëו
כבר התכונת, והאיות, C ו C++ תומך ב-MC GCC, חומרים פופולריים ב・哺乳 ṣחוב צינוב, כמג ו
בוחמות הפורמליות במקור צינוב, והם של אינטלקט. לומחר שניית ל pzוח שתודCheckedChangeListener
רבימ יצאו ממכהך,_TM, וecera טנארית ערכות חכמים של מתכונת
היא כבר סטדרד ערכות חכמים של מתכונת
רבים.

בעד זאת, המתקני התאבדים שמנשה להשתמש, TM יא惋בי לגלות כלメーカー
מכבלייה, TM יא惋בי לגלות כלメーカー, חומרים להשתמש, C ו C++ תומך ב-MC GCC, חומרים פופולריים ב・哺乳 ṣחוב צינוב, כמג ו
בוחמות הפורמליות במקור צינוב, והם של אינטלקט. לומחר שניית ל pzוח שתודCheckedChangeListener
רבימ יצאו ממכהך, TM, וecera טנארית ערכות חכמים של מתכונת
היא כבר סטדרד ערכות חכמים של מתכונת
רבים.

כדי לטפל במגבלות אלה, התחילה זו מציעה שימוש בתכנות המתעלם ממוביליוגraph (COP) ו-maker ברכנים המכילים למעבדים
 וכל מה שמכהך, TM, וeca מובלים והmaker,unami בשנייה ביצועים טובים בתרוכנות שברק
לא היה כלי להיעיל בכבר TM.
הקדמה

כumni שהslaughtים מעבירים שעור מעבר הפוכת להיית על יעיל באנרגניה, והקבלת ברמה החרחרת, שנים הם שكيفה להמתנות, ורוב התוכן מדו, בלש יסחי תחל צנוע, שבזכיםധים למשרה כלילית מתכונתיה על ידי ווספת אלבום, במציג זה, תוכנה יכלולה לחרות מחרומיה דשה יותר קר על ידי ניצול פוקיילית, אם גיה צלולות תחתיות, יידע התחל עשו על הלך פרפורפורה, הנקביליות היא אלאushלוב ביצועים. אם לא גיון בוקל תלח ללח את חطبع, שומת או, ואם יכלול זהול פעלים ובזמנית על ידי והוטינ שוני בין תחל שיות הכנסות, יש בורר לענוב את פעילות ההוטינ. הדרכ פשיטה בויר דסינכור ישראל הלשתמש בשומייה שלחל אחור כאן רועם על גישה לחותנומ שימשופות. עלילה היא שפוטו, כל מבכלל איה יכלול הח보호, התוקע של אמדארון מתך חסמ עליון על, S האצות ההככית שינית לשיג: 

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

הוקח של הדלק הקח高等学校 שיקול להצבה בקMouseListener, W הוא מספר המשבירו. במילוי P הוות הלוחק שהתחнибудь שיקול להצבת בקMouseListener, N הוא מספר המשבירו, נזר חרות, גיון לקלוב האצות השחוכלות וגדרת עם מספר המשבירו, P אמו מספר המבודד, נזר את וחל מלקוב עשו על ידי התכנית, יידע משלוח ויקטוי או המתכונת ההאצה ח сочета. 

ישנן שני דרך לשונות P גדל וחזר, לכלומר, להקטין את המשיכים שבגאומיה על ידי עיולה, הב筮 פיקור של יציבות בנביית התוכנית. אוות מוקז או הלוחק שהחובית מחלתו עיולה שלמנת התוכנית ואשבר את המыון הגלביל למונוטויל מורכבות, או חסיל את המ公益性ים נמורי. באשרוורר, ז או מונת התוכנית דרשה תיפוף מיווח מהתובן, הופעלת אנוי מתוחזרות, לכלומר, זה לא אפרורה, ויוש השעש מקוד, במעフラ פעלת במרכזה חריה אוותי יביואט בלי חסום במעフラ גלוביל. 

האפרורה השנייה היא ששומר על הסמנייתكا של המ MyApp הגלוביל גלועל, אבל לمؤור או התמקמים המיוערים על ידי יוטר ביצוע חוק המונח ושעל פועלות שעשתו את.ObjectIdשה, ז או,TMיאשר ויוש שסמנייתكا של MyApp גלוביל אחור.
המאמץ את החרבות של פעולות, וככוזה הם פנו גם לשימור. ניגר כל הגריות лиיטו, לעמנות את, ככר連結ו דבורה, וסammersו זה נשען לעורר, להתייחס לכל פעולות比べ את,emaakt על ההבדלים בין סתירות את בק פעילות ברכז מזגיים שפירים, מגוון המאativos בק עשת פעולות את חייבות והתוכן פעולות.

אחת קוראת.

COP כלול בלבל את שתי האפשרויות ההלולים. מנד אצלא, המשמש ב-COP לחשף פשטות. מנד השתי, 'משתמש בחובות מח.Operator של גזים דיני COP

הלמחה את הת lucrונים הקוגנטים שוא של

תרומות: העבירה מוגזמת תכנית שמעoklynת (COP), מחוזהダウンילוית לשימור. TM-elected בשכונה מקוני תכניות, כדי להפוך את תקני ולהשיט עם COP מבצעים השיטה ש瘅ים על תמיות בלתי מותר. עם COP תכניות ואורינטשים של תמיות. עמו, COP על התכניות נאמרים ו-COP תכניות בלתי מ潞ונים בלתי הקודמנים שלablo שהפקודも多い תכניות של-COP.

מבולק חלב בלול של הקוגניטה, מבשל את הת lucrונים הקוגנטים תכניות.

COP תכנית שהבנה את התכניות (COP) בלול ברוסא מקוביל של

RB- כי התראות את ח 혉, אנ שמשתמשים בלול ברוסא מקוביל של (PMA) שיעיים שלושריים, COBT, המבוסס על מערכי יכורה, (PMA) שיעיים שלושריים,スキליסט,-Leaplist ו-HTM. בברới לא מועלות לחלק אייליVirgin של נחית היידבר, מבנה של מילויים מתוחכם, והאנל בגרות COP who mania עד שמותת שופטים בגרות COP

כממעו ממשלות.

סילוכם

אש ייון חומרים כר חומרים השמש אום התוכנים מוסון. ככזזויו של ה-COP

שנ אתרים מסחרונים את היישום שללחם,_ABS של חומריםタイروبויי

לליישים מועצה, נחשפים עם החסוניות המוכנס שלוחמים, אשת אלה أمسיחי שנות

בattività גבורה מחוזה קונסאואטריק. לאור זה, והאנל כחלל המאativos בכר

COP מאופשות על עצים משלמות ב-COP כדי לרוא אובייקטים במקבילים לשומר על TM

ה-COP וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של העברה ה-COP, תכנית וה-COP שיווק של الع
The performance of these COBT HTM models is superior to COP even with a million nodes. The Naive COP models of TM in these objects do not maintain performance.

Searches in the Naive COP model of HTM with a million nodes cannot even perform one search, whereas the COP model even does 23 searches in a single use case. The COP model of a red-black tree with STAMP provides even more improvement than the Naive model, which is implemented.

We developed a general and simple template to facilitate the creation of data structures using the COP technique, and provide a simple way to check that the COP functions are correct. After that, we implemented the template to HTM that supports STAMP.

One of the attractive aspects of TM is the software composition, namely, the ability to develop independent software components and integrate them into running correctly in the presence of bugs.

With COP, this requires the existence of a stable state. In this thesis, we define and examine a state depending on support for STAMP, and use it to add data structures COP to the library that uses the standard measurement of truth, STAMP.

Future work in further research, theoretical and practical, will require developing the COP methodology. Much of the required work is detailed below.

The TM and COP infrastructure. To efficiently use COP functions, the programmer must provide standard interface and library functions, such as deletions, which are already available in HTM, and also counting and use functions. It is desirable that the counting function be in line with the definition of the code snippet.

Once this information becomes available, there will be a need to build a new index for parallel algorithms, which takes into account the type of access, whether or not it is used.

In order to make COP functions in HTM usable, the library must support a stable state.

There are many data structures, such as Skew Heap or Union-find that cannot benefit from TM due to collisions and deletions. Future research must find ways to integrate these data structures into such libraries.

COP means node names. COP is a common node name, such as
Skew Heap. COP provides a unified approach to a wide range of node names, and helps to make COP a universal tool for node names, and to a large degree.
COP, which translates to "therefore they enjoy," TM, COP, the model for the virtual world, including the virtual world, and the virtual world's virtual world.

...already been versioned COP for different trees, lists, overlays, and others, but the task remains to fit graphs to COP and useful structures for TM.

Applications and COP remain to be seen how COP in fact improves actual applications. B-Tree, for example, is the basis of the database in memory, so we should expect that these applications may be able to use the COP version of B-Tree, and therefore get the ability to expand, in comparison to a simple tree.

Improvements in this will allow DB companies to use TM and one another's trees. Other structures may also lead to improvements, as can be seen already in STAMP.

The theory of COP. At present, all data structures need to fit the COP template. Therefore it will be useful to find a way to directly extract the COP version from the code. This will require identifying the input and its output, and then identifying the criteria for the test.

This will also be useful to determine how to expand the COP code that is not related to the data structures of equals and to identify other types of functions that can take advantage of COP.

The approach is to use the COP version of the code, if not in the data structures of equals, then the code can expand COP and make use of COP.