Transactional Lock Elision Meets Combining

Alex Kogan
alex.kogan@oracle.com
Oracle Labs
Burlington, MA, USA

Yossi Lev
yossi.lev@oracle.com
Oracle Labs
Burlington, MA, USA

ABSTRACT
Flat combining (FC) and transactional lock elision (TLE) are two techniques that facilitate efficient multi-thread access to a sequentially implemented data structure protected by a lock. FC allows threads to delegate their operations to another (combiner) thread, and benefit from executing multiple operations by that thread under the lock through combining and elimination optimizations tailored to the specific data structure. TLE employs hardware transactional memory (HTM) that allows multiple threads to apply their operations concurrently as long as they do not conflict. This paper explores how these two radically different techniques can complement one another, and introduces the HTM-assisted Combining Framework (HCF). HCF leverages HTM to allow multiple combiners to run concurrently with each other, as well as with other, non-combiner threads. This makes HCF a good fit for data structures and workloads in which some operations may conflict with each other while others may run concurrently without conflicts. HCF achieves all that with changes to the sequential code similar to those required by TLE and FC, and in particular, without requiring the programmer to reason about concurrency.

KEYWORDS
hardware transactional memory; lock elision; flat combining; elimination; concurrent data structures

ACM Reference format:
https://doi.org/http://dx.doi.org/10.1145/3087801.3087838

1 INTRODUCTION
Flat combining (FC) [11] and transactional lock elision (TLE) [5] are two synchronization techniques that, given a sequentially implemented data structure and a lock, enable efficient application of concurrent operations to the data structure.

FC allows threads to delegate their operations to another thread, a combiner, which applies those operations while holding the lock. The key benefit of having a combiner to run operations on behalf of other threads is the ability to combine and eliminate multiple delegated operations with each other, thus reducing the total work under the lock. The code to combine and eliminate operations varies between one data structure to another. Thus, FC relies on exploiting data structure specific semantics to achieve better performance when multiple concurrent operations are applied to the data structure by the same thread. Furthermore, even when combining and elimination is not applicable, applying multiple operations by the same thread also often results in a more cache-friendly execution. FC is compelling because it relieves programmers from the need to reason about concurrency: as long as they provide a sequential code to efficiently combine and/or eliminate multiple operations together, a multi-thread execution can provide a significant performance benefit despite the fact that all operations are executed under a single lock.

TLE also introduces parallelism to lock-based sequential code; it achieves that by using hardware transactional memory (HTM), a feature introduced in recent multicore processors that allows running multiple critical sections speculatively in parallel. With TLE, operations are logically sequentialized by a critical section’s lock, but in practice they can run in parallel as long as they do not conflict on accesses to shared locations. If the speculative execution fails (perhaps after multiple trials), TLE reverts to execute the critical section under the lock.

TLE can be used to introduce concurrency between operations of a sequential data structure, and provides excellent scalability when the operations rarely conflict [15]. Unfortunately, it performs poorly in the presence of conflicts (even between a small faction of operations), which lead to lock acquisitions, as no operation can run using a hardware transaction while the lock is being held [1, 4, 7, 15]. Unlike FC, TLE does not delegate operations to other threads, and does not benefit from data structure specific optimizations that can make the execution of multiple operations under the lock more efficient.

In this paper we explore how these two techniques can complement each other. Consider, for example, a priority queue with two operations: Insert and RemoveMin. Depending on the underlying data structure (e.g., if a priority queue is based on a skip list), Insert operations may be able to run concurrently using hardware transactions, as they would access a disjoint set of memory locations. RemoveMin operations, however, will always conflict with each other, as they are all required to remove the same element from the queue. Using TLE to execute priority queue operations will therefore result in frequent failures to the lock when a few threads try to execute RemoveMin operations concurrently. At the same time, although RemoveMin operations can easily be combined together, Insert operations generally cannot, undermining the efficiency of FC in this case. Moreover, FC does not allow those Insert operations to proceed in parallel with combining of RemoveMin operations.
Note that a simple idea of applying TLE and then combining operations that fail to the lock would not address this issue either. This is because the combined RemoveMin operations would run while holding the lock, blocking all other operations, including Insert operations that do not conflict and could run in parallel.

The key contribution of this paper is the introduction of a novel approach that achieves “the best of both worlds” by using HTM to combine and run multiple delegated operations together while still allowing other operations to run concurrently with them. We implement this approach using HTM-assisted Combining Framework (HCF), a new framework that provides the following benefits:

- Works with sequential code of a data structure.
- Supports delegation, i.e., a thread can run operations on behalf of other threads.
- Delegated operations can be applied concurrently with other, non-delegated operations that are ran by their respective threads.
- Allows concurrent execution of different subsets of delegated operations.
- Allows taking advantage of data structure specific knowledge by using:
  - Combining and elimination of multiple delegated operations, resulting in a shorter total execution time.
  - Using operation specific policies to decide, at run time, whether (or when) an operation should be delegated.

To elaborate on those points, HCF does not require a thread to decide upfront whether to delegate an operation or execute it by its own. A thread can start by trying to execute an operation speculatively just like with TLE, and only after a certain number of failures make it available for other threads to execute it on its behalf. Furthermore, using HTM, multiple threads can run their own operations concurrently with multiple combining threads, each applying a separate set of delegated operations.

Note that HCF is not intended to improve over FC or TLE when one of those is beneficial on its own, i.e., where operations cannot execute in parallel on HTM due to conflicts, but may combine and/or eliminate one another (the case where FC may perform well), or where all operations can run in parallel using HTM (the case where TLE is beneficial). Rather, HCF is aimed for data structures and workloads that fall in between these two cases, where some of the operations can run efficiently in parallel, and other cannot, but may benefit from combining. Later in the paper we describe a few examples for such data structures and workloads, and show how they benefit from the properties of HCF.

2 THE HCF FRAMEWORK

2.1 Overview

HCF executes operations of a sequentially implemented data structure protected by a lock. An execution of an operation invoked by a thread \( T \), denoted as the operation’s owner, goes through at most four phases until completion:

(1) \texttt{TryPrivate}: the owner tries, perhaps several times, to execute the operation using a hardware (HW) transaction.

(2) \texttt{TryVisible}: the owner announces its operation by adding it to a publication array, a container holding operations that are visible to other threads, and then tries, perhaps serval times, to execute them using HW transactions.

(3) \texttt{TryCombining}: a thread selects a subset of the announced operations, including its own, and executes them using one or more HW transactions. In this phase we can benefit from data structure semantics to combine and/or eliminate multiple operations, as well as to adjust the number of operations executed by a single HW transaction.

(4) \texttt{CombineUnderLock}: the combiner acquires the data structure lock and executes, without using HW transactions, the operations that were selected but not completed in the \texttt{TryCombining} phase.

An operation returns once one of these phases is completed successfully, as demonstrated below:

\begin{verbatim}
Execute(Op) {
    // Choose the publication array \( Op \) is associated with
    Pa = choosePubArr(Op);
    if (!TryPrivate(Op,Pa) &&
        !TryVisible(Op,Pa) &&
        !TryCombining(Op,Pa)) {
        CombineUnderLock(Op, Pa);
    }
    return OpRetVal;
}
\end{verbatim}

As suggested by the pseudocode, there could be multiple publication arrays, where each operation may reside in only one of them at any given time. Supporting multiple publication arrays, and configuring how HCF handles operations using each of those arrays (e.g., how many HTM attempts are made in each of the first three phases) is a key to customizing HCF for each particular operation. Going back to the priority queue mentioned in the Introduction, RemoveMin operations can use a separate publication array from Insert operations. This way, we can configure HCF to skip HTM attempts in the first two phases for (highly contended) RemoveMin operations and go directly to the combining phases (TryCombining and CombineUnderLock), after announcing the operation in TryVisible. On the other hand, Insert operations (using a separate publication array) can be handled differently, trying HTM attempts in all of the first three phases.

Critically, the configuration of HCF, including the number of publication arrays and the assignment of operations to these arrays, cannot affect the correctness, but only the performance. In particular, each operation is guaranteed to execute exactly once (note that once an operation is announced, it may be applied in either the TryVisible phase by its owner, or in the TryCombining or CombineUnderLock phases by its owner or by a combiner). Using HW transactions (in configurations that enable them) is crucial for providing this guarantee. We elaborate on this point in the following sections.

2.2 Algorithmic Details

Like with FC, each operation is associated with an operation descriptor that contains input arguments required for the sequential execution of the operation and a field to store the operation result (when applicable). In addition, the descriptor includes a status field that
is used to synchronize between the different phases, and holds one of the following values: UnAnnounced, Announced, BeingHelped, or Done. The field is initialized to UnAnnounced when an operation is invoked.

The operation descriptor also specifies three sequential methods: runSeq, shouldHelp and runMulti. The runSeq method is used for running the operation's sequential code. The shouldHelp method is used by a combiner to select a subset of (announced) operations from a publication array (to execute in the TryCombining and CombineUnderLock phases). The runMulti method is used to combine and eliminate multiple operations with one another, and then apply the combined operation(s) on the data structure. The runSeq and runMulti can be sequential simply because they are always executed either while holding the data structure lock, or using a HW transaction. On the other hand, allowing the shouldHelp to be sequential requires some additional care, for which we use an additional selection lock, as we later describe when presenting the HCF framework pseudo code. All the three methods implement only the logic of the selection, combining, and application of the data structure operations, and are independent of the operation descriptor usage by the HCF algorithm to synchronize between the different phases.

runSeq is the only method that the programmer is required to provide (which can be just a wrapper around a sequential implementation of the operation); the framework provides multiple default implementations for shouldHelp and runMulti. For example, one default implementation for shouldHelp returns true for every invocation, causing the combiner to choose all announced operations in the publication array, while another variant returns false, causing the combiner to apply only its own operation. The former is useful when all operations in a particular publication array can be easily combined, while the latter when combining is not applicable and we rather run each operation by its own (in which case runMulti does not really help any other operation). Similarly, the framework provides a runMulti implementation that simply runs the chosen operations, without combining them, by invoking runSeq for each. The user can override those default implementations and provide variants tailored for a specific data structure, e.g., shouldHelp that selects only operations that can be combined with each other, and runMulti that combines and runs them as a single operation. Yet, even the default generic implementations can be useful, e.g., when used with publication arrays that only hold operations that cannot benefit from combining.

We now describe the main functions for each of the algorithm phases mentioned in Section 2.1: the pseudo-code is shown in Figure 1. We note that the HTMClause-onAbort syntax used in Figure 1 is merely a convenient shortcut for the standard syntax of:

```java
if (begin hardware transaction) {
    transaction code;
    commit;
} else { abort handling code; }
```

The TryPrivate function, implementing the corresponding TryPrivate phase, acts on an operation that is still in its UnAnnounced state, and simply tries to execute it using a HW transaction, up to TryPrivateTrials times. Like with TLE, each HW transaction subscribes to the data structure’s lock L, and aborts if it is acquired (Line 5). If the operation was not completed by TryPrivate, we move to the TryVisible phase, implemented by the corresponding TryVisible function. The latter begins by setting the operation descriptor status to Announced, and then adds it to the publication array Pa (Line 14). It then continues to try executing the operation using a HW transaction, up to TryVisibleTrials times, similar to the TryPrivate function. In this case, however, because the operation is now visible to a combiner thread, some care must be taken to ensure that the operation is not executed twice — once by the owner and once by a combiner thread.

The approach we take is as follows. First, we check that the status of our operation remains Announced when starting a HW transaction (Line 18). Second, we disallow applying our operation while another (combining) thread is selecting operations it will help to, so we will not execute an operation that is selected by that thread. This is achieved by checking the selection lock (Line 19) when starting a HW transaction; this lock is held throughout the selection process as described below. Finally, we remove the announced operation from the publication array in the same transaction that has applied this operation (Line 22). This is done to avoid the race between the owner of the operation and a combiner thread that starts selection of operations after the owner has applied its operation and before it has removed that operation from the publication array.

If the transaction aborts and the operation is no longer in the Announced state, then a combiner is responsible for executing this operation. Thus, the owner waits for the combiner to complete the operation by spinning on the status field, waiting for it to be changed to Done (Line 27).

Next, if the operation did not complete in the TryVisible phase (and was not chosen to be executed by a combiner), its owner thread T moves on to the TryCombining phase implemented by the corresponding TryCombining function. There, it tries to become a combiner for the operations in Pa. It does so by acquiring the selection lock of the Pa that ensures that it is the only thread that selects operations from that Pa. Before T continues with the selection of operations from Pa, though, it has to check that while competing for the selection lock, its own operation was not selected (and, perhaps, applied) by another thread that acquired the selection lock before T. If T’s operation was selected, T immediately releases the selection lock and waits for the status of its operation to become Done (cf. Line 40). Otherwise, it is guaranteed that T’s operation status remains Announced, because no other thread can choose T’s operation that resides in Pa while T is holding Pa’s selection lock. At that point, T calls the auxiliary chooseOpsToHelp method to choose the subset of operations from Pa to apply (Line 43). chooseOpsToHelp (not shown for simplicity) scans the publication array, calling shouldHelp for every announced operation it finds there (except for T’s operation, which is chosen by default). If an operation is chosen (i.e., the corresponding call to shouldHelp returns true or it is T’s operation), its status is changed by chooseOpsToHelp to BeingHelped, it is removed from the publication array, and is added to the set returned by chooseOpsToHelp.

Note that when scanning a publication array, chooseOpsToHelp does not need to see a consistent snapshot. In fact, new operations

---

1 In our implementation we simply use an array with a slot per thread, but other dynamic schemes are possible.
Trials
if 18
12
11
} { 24
23
return true
Pa.remove(Op);
// remove Op from Pa as part of the HW Tx
Op.runSeq();
Pa. selectionL .isLocked ()
19
op. Status = Announced;
13
} { 33
32
31
}
30
numFailures++; 28
27
// a combiner helped or is helping us
26
while (Op.status == BeingHelped) yield ();
25
return true;
24 }
23
} 22
if (Op.status != Announced)
21
continue;
20
L.unlock ();
19
opsToHelp.remove(opsHelped);
18
while (Op.status == BeingHelped) yield ();
17
return true;
16 }
15
while (numFailures < TryVisibleTrials) { 14
13
numFailures = 0;
12
Pa.add(Op);
11 }
10
} 9
8
return true
6
Op.runSeq ();
5
HTMClause
4 }
3
if (L.isLocked()) abortHT;
2
Op.runSeq();
1 return true;
0 }
TryPrivate(TryPrivateTrials)
{ 2
numFailures = 0;
1 }
TryVisible(TryVisibleTrials)
{ 12
op. Status = Announced;
11
Pa.add(Op);
10 numFailures = 0;
9 }
8 while (numFailures < TryVisibleTrials) {
7
HTMClause
6 }
5
if (L.isLocked()) || Op.status != Announced ||
4
Pa. selectionL .isLocked () || Op.status != Announced ||
3
numFailures++;
2
} 1
return false;
0 }
TryCombining(TryCombiningTrials)
{ 35
Pa. selectionL .lock ();
34
}
33 if (Op.status != Announced)
32
if (numFailures < TryPrivateTrials) {
31
HTMClause
30 }
29 if (L.isLocked()) abortHT;
28
opsHelped = Op.runMulti(opsToHelp);
27 }
26 onAbort { numFailures++; continue; }
25 }
24 opsToHelp = Op.chooseOpsToHelp(Pa);
23 Pa. selectionL .unlock ();
22 numFailures = 0;
21 while (numFailures < TryCombiningTrials) {
20
HTMClause
19 }
18 if (L.isLocked()) abortHT;
17
opsHelped = Op.runMulti(opsToHelp);
16 }
15 onAbort { numFailures++; continue; }
14 }
13 opsToHelp = Op.chooseOpsToHelp(Pa);
12 Pa. selectionL .unlock ();
11 numFailures = 0;
10 while (numFailures < TryCombiningTrials) {
9
HTMClause
8 }
7
if (L.isLocked()) abortHT;
6
opsHelped = Op.runMulti(opsToHelp);
5 }
4 onAbort { numFailures++; continue; }
3 }
2 opsToHelp = Op.chooseOpsToHelp(Pa);
1 Pa. selectionL .unlock ();
0 numFailures = 0;

Figure 1: HCF framework functions. Code in HTMClause executes in a hardware transaction; if aborted, control is transferred to the onAbort clause. A return statement inside HTMClause commits the transaction before returning from a function.

can be added to the array while the combiner scans it, but they cannot be removed (by their owners) since T is holding the Pa’s selection lock. This allows an efficient selection process without atomic operations.

Once chooseOpsToHelp returns the set of selected operation descriptors, stored in opsToHelp. The TryCombining function repeatedly calls runMulti to execute some (or all) of the selected operations. Recall that runMulti is a sequential function and assumes that it runs in isolation, and therefore it must be called from within a hardware transaction (or under a lock). If a call to runMulti completes successfully within a hardware transaction, the status field of operations executed by that runMulti call is set to Done (Line 52), signaling the owners of those operations that the operations have been completed. After that, these operations are removed from the opsToHelp set (Line 53). We invoke runMulti multiple times to allow an implementation where it executes only some of the selected operations at each call, so they are more likely to complete by a single hardware transaction.

The TryCombining function completes if either it successfully executed all selected operations, or if all budgeted (TryCombining-Trials) HW transactions failed while trying to execute them. In the former case, opsToHelp is empty, and TryCombining returns true (Line 54). Otherwise, T moves on to the CombineUnderLock phase, implemented by the corresponding CombineUnderLock function. The latter acquires the data structure lock L (Line 59), and executes the rest of the operations in opsToHelp by repeatedly calling runMulti. This is similar to the TryCombining function, except that now runMulti is called outside of a hardware transaction. Once opsToHelp is empty (that is, all selected operations were executed successfully), the CombineUnderLock function releases L (Line 65) and completes.

2.3 Correctness
In this section we outline the main claims showing that using HCF with a sequential implementation of a data structure results in a correct, linearizable concurrent implementation of the data structure.

First, if an operation is applied to a data structure via the HCF framework, then it is done by either calling the runSeq or the runMulti method (that may call the runSeq method for that operation, or apply that operation as a part of another, combined operation). In any case, this is done either while holding the data
structure lock, or inside a hardware transaction. Thus, since no hardware transaction can access the data structure while the lock is held, the operation seems to take effect atomically, either when the transaction commits or when the lock is released.

Second, we note that every operation always takes effect between the operation’s invocation and response. This is trivial if the operation is applied by its owner. When the operation is applied by a combiner, the combiner learns about it only after the operation has been invoked (and announced by its owner). Along with that, the combiner changes the status of the operation to Done only after the operation takes effect. The owner of the operation returns a response only after it learns that the operation’s status was changed to Done. Thus, even when the operation is applied by a combiner, the response is returned only after the operation takes effect.

Next, we claim that an operation is never applied more than once by the HCF framework. Recall that each announced operation has exactly one descriptor, and it is associated with exactly one publication array. Thus, it is sufficient to show that for any pair of Op and Pa, no series of invocations of the HCF methods can result in multiple applications of Op. To this end, consider the first invocation of runSeq (or runMulti) with Op that completed successfully (i.e., it was executed by a committed hardware transaction, or while holding the lock L). If that invocation was done by the TryPrivate function, then Op is not yet stored in the Pa at this point, and thus could not be executed by any other thread running as a combiner. Furthermore, TryPrivate returns true in this case. Thus, no other HCF function is called for Op, so Op was applied exactly once.

If the first successful invocation was done by the TryVisible function, then we know that: a) The operation descriptor status field had the Announced value when the operation took effect. b) A combiner has not been executing chooseOpsToHelp. c) The operation descriptor was removed from Pa as part of the transaction that applied Op. The fact that the status of the operation was still Announced when it was applied implies that a combiner thread in the TryCombining function has yet changed its status to BeingHelped. The fact that no combiner has been executing chooseOpsToHelp means that no other thread was reading the publication array during the execution of Op. The fact that we successfully removed the descriptor from Pa in the hardware transaction implies that any subsequent traversal of the publication array in chooseOpsToHelp will not find Op, and thus no combiner will try to apply Op again. Thus, the operation Op is applied exactly once (by its owner). The TryVisible function then returns true, and thus no additional attempts to execute Op are done.

Similarly, the TryCombining function only executes Op if it first changes its status to BeingHelped, which can only happen if the descriptor is still in the publication array when it executes chooseOpsToHelp. Thus, if Op was applied by the TryCombining function, it must be in a BeingHelped state, and hence can no longer be applied by the TryVisible function. Furthermore, upon selection of Op in chooseOpsToHelp, TryCombining removes it from Pa so any other combiner could select (and apply) it as well. We note that unlike the TryPrivate and TryVisible functions, TryCombining may return false even if Op was successfully completed; however, because the descriptor of Op in that case is no longer in the opsToHelp array, combineUnderLock will not execute Op again.

Finally, the CombineUnderLock function runs after acquiring L. This disallows the owner of Op (as well as any other thread) to make any progress until L is released or Op is completed (unless the owner of Op happens to be the combiner). Once the combiner applies Op in CombineUnderLock, it changes its status to Done. Since the owner of Op checks the status of Op in a HW transaction in TryVisible and after acquiring the selection lock in TryCombining, it will not attempt to apply Op again.

Progress properties: Unlike the original FC algorithm [11], not all usage patterns of the HCF framework result in a starvation free algorithm. In particular, because we may have multiple publication arrays, an unfair lock L may have a combiner thread executing CombineUnderLock on one array starve another combiner thread running on a different array. Since a descriptor only resides in one of the arrays, this may cause an operation to be starved if L is unfair.

If L and selection locks guarantee starvation freedom, however, a concurrent implementation of a data structure that uses the HCF framework will also be starvation free, that is, each operation will eventually complete as long as a thread that is holding the lock keeps taking steps and eventually releases it. To see why, note that with starvation free locks, any thread that calls TryCombining(Op,Pa) will eventually complete Op. This is because it will either discover that Op was applied by another combiner or it will select Op in chooseOpsToHelp (chooseOpsToHelp is assumed to select at least the operation of the thread that executes it). In the latter case, HCF guarantees that the combiner returns only after applying all operations it has selected by chooseOpsToHelp, including Op. The only case in which the owner of an incomplete operation may be prevented from reaching the TryCombining function is if it spins indefinitely in the loop at Line 27. However, as just discussed, an operation in a BeingHelped state will be helped by the combiner thread that transitioned it to that state, and that combiner must eventually complete its operation given that the locks are starvation free.

2.4 Framework Customization

HCF supports two mechanisms in order to allow parallel combining. First, we can use one publication array for all operations and have chooseOpsToHelp choose only operations that would easily combine with the operation of the thread running it. For instance, in the case of a set based on a binary search tree, one can choose to combine only operations on the same key or on keys in the same sub-tree. Another mechanism is having multiple publication arrays with separate combiners that always select all operations announced in the respective array. Such approach is appealing when it is known a-priori which operations are expected to conflict with each other, e.g., operations on different ends of a double-ended queue. Note that in this case, we do not need to allow multiple combiners for the same publication array. In fact, we have created a specialized version of our framework for this case. There we let the combiner to hold the selection lock of the publication array for the whole duration of its execution (rather than just for the selection process). This prevents any operation in the TryVisible phase from running when there is an active combiner for the same publication array (and thus avoids conflicts between those operations with the combiner). At the same time, it avoids the need to use an
extra state (BeingHelped) and thus simplifies the implementation of the framework.

We note that those two mechanisms are not mutually exclusive. That is, we may configure HCF to use several publication arrays with chooseOpsToHelp that chooses a subset of operations to be combined, or we can use one publication array and have the combiner thread hold the selection lock until it is done. The latter case provides a form of contention control (since selected operations cannot run concurrently with the combiner), which in some sense is similar to the idea of using an auxiliary lock in TLE [1]. The contention control in HCF is more efficient, though, since by running multiple operations by the same combiner thread we reduce the contention on the selection (i.e., auxiliary) lock. Besides, by allowing the combiner to run several operations in the same hardware transaction, the overhead of starting and committing transactions is amortized.

The flexibility of HCF allows straightforward implementation of other synchronization techniques on top of it. For instance, TLE is achieved when the number of HTM attempts in the second and third phases are set to 0, while chooseOpsToHelp returns only the operation of the combiner (which then applies it under the lock). The functionality of FC is achieved when the number of HTM attempts in all the first three phases is set to 0, while chooseOpsToHelp returns all announced operations. In general, HCF allows customization not only for a particular data structure, but also divergent customizations for different operations of the same data structure. Moreover, because the number of publication arrays used as well as the way operations are assigned to the arrays is the issue of performance, not correctness, the customization may be dynamic — we can begin with a certain number of publication arrays and the way operations are assigned to them, and change that on-the-fly to better fit the given workload. It is fair to assume that no single configuration of HCF fits all data structures and workloads, calling for an adaptive runtime mechanism to tune the HCF performance. Exploring such a mechanism is left for future work.

3 PERFORMANCE EVALUATION

3.1 HCF vs. FC and TLE

We start by a qualitative comparison of HCF to the original FC technique [11]. Concurrent threads make most use of HTM when they do not conflict with each other; when data conflicts are frequent, time spent on futile HTM attempts is wasted. As a result, one should not expect HCF always to be the winner when the contention is high, e.g., when experimenting with a stack or when a priority queue is used in a workload composed solely of RemoveMin operations.

We expect the benefit of HCF over FC, however, to be evident when a data structure does allow at least some amount of parallelism. The HCF algorithm is designed to exploit this parallelism in multiple dimensions. First, it allows threads to access the shared data structure concurrently without any help from a combiner. Second, even when a thread fails to complete its operation on its own and decides to become a combiner, it allows other threads, i.e., other combiners and non-combiners, to proceed concurrently. As a result, HCF is likely to provide substantial benefit over FC in a wide range of data structures and workloads, e.g., in hash tables, search trees, skip lists, etc.

When comparing HCF to TLE [5], we note that differences in performance results are likely to arise only in workloads that involve contention. Otherwise, when threads apply their operations with one (or a few) HTM attempt(s), the techniques are similar and expected to deliver the same performance. Under contention, however, HCF has a few important advantages over TLE. First, it enables combining and elimination, potentially reducing the total amount of work required to apply contended operations. Second, HCF provides better control of the level of allowed parallelism between threads applying contending operations. In particular, while TLE allows either all threads to run in parallel or only one thread (that holds the lock) to run [5], HCF can gradually reduce the level of parallelism by having one combiner help some operations (whose owners are not allowed to run concurrently with it), while other threads can still run their operations concurrently. Finally, HCF reduces contention on the global lock that protects the access to the shared data structure. This is because the largest number of threads competing over this lock is equal to the number of combiners in HCF as opposed to the total number of threads in TLE. Thus, while the actual benefit of HCF over TLE depends on the number of combiners and the potential for the combining and elimination optimizations, we expect this benefit to increase with the contention level, leading to better scalability for HCF.

3.2 Experimental Setup

We implemented the HCF framework in C++, using templates for customizing the framework for different data structures and synchronization techniques, as we describe in the following sections.

All experiments were run on an Oracle Server X5-2 (dual socket, each socket featuring an Intel Xeon E5-2699 v3 processor with 18 hyper-threaded cores per socket) running at 2.3GHz and operated by Ubuntu 16.04. In order to reduce the effect of non-uniform memory access (NUMA) on the results, we used only one socket in this machine. We experimented with both sockets as well, and observed the performance of all algorithms being harmed by inter-socket communication when threads were running on both sockets, consistent with observations made in [2]. The relative performance, however, has not changed from the one observed when only one socket was used; we show a representative example of this behavior in one of the charts below. To eliminate the impact of non-deterministic thread placement and thread migration, all threads were pinned to cores s.t. thread i and i + X were sharing the same core (where X=18 is the number of cores per socket).

In each experiment, threads accessed the data structure concurrently and independently, applying operations selected uniformly and randomly from the given workload distribution. We report the average total throughput based on five runs per each configuration. We note that the standard deviation of those results is small, in the order of a few percents or less from the mean for the vast majority of the results and up to 9.5% in the worst case (21.1% in the runs that use both sockets).

3.3 Hash Table

We have implemented a simple sequential hash table data structure which contains a preset number of buckets. Each bucket is a singly linked list of key-value pairs whose key hashes into the bucket. In
addition, in order to support efficient iteration over key-value pairs stored in the table, we also maintain a doubly linked list of those pairs (which we call a table list). Thus, the Insert operation on the table locates the bucket into which the given key should be inserted by hashing the key, and scans the list of elements in the bucket (which is expected to be very short). If the key is already there, it completes by updating the value. Otherwise, it allocates a new node, initializes it with the given key and value, and inserts that node into the head of the bucket list as well as into the head of the table list. The hash table also supports Find and Remove operations with usual semantics, except that Remove also removes the given key from the table list.

Given this sequential implementation, we note that for any reasonable table capacity, Find and Remove operations are unlikely to conflict when run concurrently as hardware transactions. This is despite the fact that Remove modifies the table list, as it removes the key from a random location in the list, without scanning the list or reading its head pointer. Insert operations, on the other hand, are expected to experience contention as they all insert into the head of the table list. The different nature of the data structure operations makes it fit naturally in the gap between TLE and FC that HCF is designed to address: having all operations run using TLE is not likely to perform well because Insert operations will tend to acquire the lock (after a few attempts on HTM), blocking Remove and Find operations that could otherwise run in parallel; on the other hand, using FC will have all operations execute while holding the lock, waisting potential parallelism between the Find and Remove operations. HCF allows to customize the hash table implementation as follows. We used one publication array for Find and Remove operations and configured their respective operation descriptors to operate like in TLE, that is, skipping HTM attempts in TryVisible and TryCombining and going directly to the CombineUnderLock phase if HTM attempts in the TryPrivate phase fail. For Insert operations, we used another publication array and configured its operation descriptor to exploit all four phases, with the default chooseOpsToHelp function that selects all announced operations in the array (in addition to the owner’s operation) and combines them into one operation on the hash table. To this end, we added a (sequential) Insert-n operation to the hash table that allows insertion of multiple key-value pairs. The advantage of this operation over simply invoking Insert n times is in its ability to chain new key-value pairs that can be inserted into the table list with just one modification of the head pointer.

In the hash table experiment, we set the key range and the number of buckets to 16K and prefill the table to the half of that range. We compared the following alternatives:

- **HCF**: the HCF-based implementation described above,
- **Lock**: simply uses a lock for every operation,
- **TLE**: implements TLE over that lock,
- **FC**: implements flat combining, in which Insert operations are combined using the same Insert-n operation described above while announced Find and Remove operations are applied sequentially afterwards,
- **SCM**: uses TLE with an auxiliary lock for conflicting transactions [1].

**TLE+FC**: employs a naive combination of TLE and FC, where a thread tries to apply an operation using HTM (as in TLE), and if fails, proceeds (as in FC) by announcing its operation and trying to become a combiner under the lock (and then using the same Insert-n operation for Insert operations).

All versions that use transactions (HCF, TLE, SCM and TLE+FC) were given the same total budget of (ten) HTM attempts. For HCF operation descriptor configuration that uses all four phases (e.g., for Insert operations), we set TryPrivateTrials, TryVisibleTrials and TryCombiningTrials to 2, 3 and 5, respectively. We found experimentally that this setup works reasonably well across a wide range of data structures and workloads. Thus, we opted to use one setup for all experiments with HCF that involve all four phases, rather than show the best choice of those parameters for each case.

Figure 2 shows results for three workloads where 40, 80 and 100% of the operations are Find, respectively, while the rest of the operations in each workload are split evenly between Insert and Remove. Note that Figure 2 (b) presents data with 72 threads, i.e., collected on both sockets of the machine. This is a representative example that shows how NUMA effects harm the performance of all algorithms. Since interesting observations happen when the NUMA effects do not present, the focus of our analysis is on the results achieved on a single socket.

As Figure 2 (a) shows, when the workload does not include any updates, HCF scales similarly to TLE (as well as SCM and TLE+FC), exhibiting no overhead comparing to those alternatives. This is not surprising, as most operations are expected to complete on HTM, without acquiring the lock. The difference in scaling for all these variants after 18 threads can be attributed to the enabled hyper-threading. Along with that, Lock and FC show no scalability as they execute all operations under the lock.
When workloads include updates (cf. Figure 2 (b) and (c)), the HCF variant peaks with higher throughput and, perhaps more importantly, preserves this throughput as the number of threads increases (modulo the effects of cross-socket communication when going beyond a single socket). This is in contrast to all other variants that employ HTM in which conflicts between Insert operations increase the rate of lock acquisitions; we provide the evidence that supports this claim below.

In general, the gap between HCF and other alternatives grows with the number of update operations. This is because more operations fail to complete on HTM due to increased number of conflicts between Insert operations. Instead of acquiring the lock and blocking all other operations, however, HCF allows more parallelism by attempting to combine those operations using HTM. Also, we note that TLE+FC performs almost identically to TLE, providing very little benefit and in particular, performing substantially worse than HCF. As we show below, this is in part because TLE+FC, in contrast to HCF, combines only a few operations in practice.

In the rest of this section, we present and discuss some additional performance data that shed some light on where the advantage of HCF over its alternatives comes from. As an example, we focus on the workload presented in Figure 2 (c), i.e., 40% Find operations. Figure 3 shows the percentage of operations completed in each of the four phases of HCF, for all operations as well as when considering Insert operations separately from Find and Remove operations. As Figure 3 shows, these two classes of operations perform differently indeed. While Find and Remove operations do not suffer from conflicts and manage to succeed on HTM even when the number of threads grows, Insert operations experience conflicts. TLE+FC produces less cache misses compared to TLE, but only slightly since, as we have seen above, the combining degree of the former is very small.

In summary, the hash table experiments show one of the main benefits of HCF, which is enabling operation specific policies. That is, conflict-free operations are executed in TLE-like fashion, enjoying the very low rate of failures to the lock. At the same time, conflicting operations employ combining (following attempts on HTM), ripping the benefits typical to FC but without compromising the parallelism of other, non-conflicting operations.

3.4 AVL Tree-based Set

The hash table example demonstrates how well HCF deals with data structures that have different type of operations: ones that rarely conflict with each other, and work well with TLE, and others that often conflict, and are better fitted to the FC technique. In this section, we focus on data structures where any of their operations may experience contention, depending on the actual workload. Unlike the case of the hash table, here the programmer cannot know upfront which (or whether) operations will experience contention, and has to rely on the ability of HCF to dynamically select conflicting
operations, and apply them together (perhaps using combining and elimination) while letting operations that do not conflict to run in parallel. We demonstrate the importance of this dynamic selection feature in HCF using experiments with an AVL tree-based set.

The AVL tree data structure is known to scale well under TLE when keys stored in the tree are accessed uniformly at random [2]. In addition, in the case of a uniform workload, when the tree is accessed through the set interface (with Find, Insert and Remove operations), the combining and elimination are known to be unprofitable [11]. However, an interesting question to consider is whether these two facts hold under skewed, non-uniform workloads.

To address this question, we used the HCF framework with the following customization. We used one publication array for all operations, for which a combiner was configured to select only pending (announced) operations on keys falling in the same (left or right) subtree (of the root) as its own operation. To this end, we modified a sequential AVL tree implementation through a few trivial changes to maintain a look-aside variable that holds the root’s key; this variable is used by chooseOpsToHelp. In the custom version of runMulti.t1, the selected operations are then sorted by their key and operation type, and applied to the tree after operations on the same key are combined and eliminated according to the semantics of the set interface (e.g., out of two or more Insert operations on the same key that does not present in the set, only the first one takes effect on the tree, while all others return with an indication that the key is already in the set).

Figure 5 shows throughput results for the AVL tree-based set experiment. There, a tree is initialized to hold half the key range of [0..1023] and threads access keys according to a Zipfian distribution with the skew parameter $\theta = 0.9$. ($\theta$ is a parameter of the distribution in the range [0..1), where a higher number gives a higher probability to the lower part of the key range, thus producing more skewed distribution). We compare the HCF variant to Lock, TLE, SCM, FC and TLE+FC, where the latter two use the same function to apply multiple operations as HCF (albeit on all announced operations).

Figures 5 (a), (b) and (c) show results for the case where 0%, 40% and 80% of operations are Find, respectively, while the rest of the operations are split evenly between Insert and Remove. As expected, we see the advantage of HCF over other alternatives to grow with the increase in the number of update operations (and thus conflicts between hardware transactions). This is because when the conflict rate is high, more operations need to announce and become (or get helped by) a combiner, providing more benefit to the combining and elimination optimizations. Those operations would normally require a lock in TLE or SCM, blocking all other threads and thus leading to sub-par performance of those variants compared to HCF.

In general, the correlation between the rate of conflicts and the advantage of HCF was notable when we experimented with different key ranges and/or different $\theta$ parameters of the Zipfian distribution. Other performance statistics, such as the combining degree of various variants and L1 Data cache miss rates, exhibit similar patterns and lead to the same conclusions as in the case of hash table. We also note that we experimented with other HCF-based variants, e.g., one that does not use combining and elimination optimizations (but rather a combiner applies all announced operations one after another) and another that uses two publication arrays, one for each of the two subtrees under the root. All alternative variants performed closely to, but worse than, the HCF variant shown in the experiments.
in the figures. This underlines the importance of two features of HCF, which are allowing a combiner to run concurrently with other operations (in this case, operations on another sub-tree, or operations on the same sub-tree that were not selected by the combiner) and supporting efficient application of multiple operations through combining and elimination.

4 RELATED WORK

The FC technique uses one global lock to protect the access to a shared data structure [11]. This key feature is what makes designing concurrent data structures that use FC trivial, and works well when the data structure has one or a few contention points. In fact, as Hendler et al. show in [11], and as demonstrated by a revised combining technique in [9], combining-bases stacks and queues perform much better than other, blocking or non-blocking alternatives. However, the use of a single lock can harm scalability when the data structure allows concurrent access.

Several papers have considered extending the FC technique to support multiple combiners. Hendler et al. [12] present the design of concurrent synchronous queues using a parallel flat combining algorithm. The idea is to split dynamically the publication array into chunks, where each chunk can be processed in parallel by a different combiner. Each combiner matches pending requests in its chunk, and uses an additional single-combiner synchronous queue to store an overflow of operations of the same type that did not have a matching pair. Budovsky attempts to use multiple combiners for designing skip-lists-based sets [3]. His idea is to divide the skip-list statically into multiple non-intersecting regions, where requests belonging to each region are managed by a different combiner. The margins of each region are identified by specially marked nodes of the skip-list; these nodes are assumed to be immutable. Unlike HCF (and the original FC technique), however, both these approaches are tailored to very specific data structures. Moreover, they seem to perform well only when operations of the data structure can eliminate each other [5, 12].

More recent work by Drachsler-Cohen and Petrank [8] considers applying combining to lazy linked lists [10], where threads traverse list nodes in a lock-free fashion, and acquire a per-node lock if they need to update that node. The idea in [8] is to modify locks used to protect list nodes so that a thread acquiring the lock may combine its operation with those of other threads waiting for the same lock. This technique inherently relies on the existence of a scalable concurrent implementation that uses locks, and thus is not intended for sequentially implemented data structures.

The TLE technique [5] (or, more precisely, the closely related SLE technique [14]) was introduced more than fifteen years ago, but it became practically useful only recently with the availability of commercial architectures featuring HTM. TLE was shown to scale nearly linearly when concurrently running threads do not conflict, but its performance deteriorates substantially when such conflicts do occur or when capacity limits are reached [1, 7, 13, 15]. Several recent papers suggest ways to enhance TLE to cope better with these limitations. Afek et al. [1], for instance, propose to use an auxiliary lock to synchronize between threads that fail to commit their respective HW transactions due to contention. Diegues et al. [6] introduce core locks, which synchronize between threads running on the same core when transactions fail due to capacity limits. To the best of our knowledge, however, none of the previous work in the context of TLE considers combining operations of concurrent threads, e.g., those that have conflicts or share the same core/socket.

5 CONCLUSIONS

In this paper, we explored how one can effectively combine advantages of the pessimistic FC technique [11] and the optimistic HTM-based TLE approach [5] into a novel synchronization framework called HCF. In particular, we discussed how the FC technique can leverage HTM to allow parallel execution of operations on sequential data structures that are protected by a global lock, while still enjoying the benefit of elimination and combining that FC provides. Using multiple examples, we demonstrated that HCF can significantly outperform TLE, FC and other related alternatives on a variety of workloads.

ACKNOWLEDGMENTS

We would like to thank Tim Harris, Erez Petrank and anonymous reviewers for insightful comments and great suggestions to improve the presentation of the material in the paper.

REFERENCES