Lock-Free Locks Revisited

Naama Ben-David
VMware Research, USA
bendavidn@vmware.com

Guy E. Blelloch
Carnegie Mellon University, USA
guyb@cs.cmu.edu

Yuanhao Wei
Carnegie Mellon University, USA
yuanhao1@cs.cmu.edu

Abstract

This paper presents a new and practical approach to lock-free locks based on helping, which allows the user to write code using fine-grained locks, but run it in a lock-free manner. Although lock-free locks have been suggested in the past, they are widely viewed as impractical, have some key limitations, and, as far as we know, have never been implemented. The paper presents some key techniques that make lock-free locks practical and more general. The most important technique is an approach to idempotence—i.e. making code that runs multiple times appear as if it ran once. The idea is based on using a shared log among processes running the same protected code. Importantly, the approach can be library based, requiring very little if any change to standard code—code just needs to use the idempotent versions of memory operations (load, store, LL/SC, allocation, free).

We have implemented a C++ library called Flock based on the ideas. Flock allows lock-based data structures to run in either lock-free or blocking (traditional locks) mode. We implemented a variety of tree and list-based data structures with Flock and compare the performance of the lock-free and blocking modes under a variety of workloads. The lock-free mode is almost as fast as blocking mode under almost all workloads, and significantly faster when threads are oversubscribed (more threads than processors). We also compare with several existing lock-based and lock-free alternatives.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Keywords: locks, lock-free data structures, idempotence, helping

1 Introduction

To be or not to be lock free, that is the question. Lock-free, or *non-blocking*, algorithms, are guaranteed to make progress even if processes fault or are delayed indefinitely. They are, however, burdened with some issues. One important issue is that they tend to be significantly more complicated than their lock-based, *blocking*, counterparts. Even basic data structures such as stacks, queues, and singly linked lists can lead to non-trivial lock-free algorithms with subtle correctness proofs. More sophisticated data structures, such as binary trees and doubly linked lists, become considerably more complicated. If one needs to atomically move data among structures, lock-free algorithms become particularly tricky. Developing efficient algorithms with fine-grained locks is not necessarily a cakewalk, but is typically significantly simpler.

Another issue is performance. The relative performance of blocking vs non-blocking algorithms depends significantly on the environment in which they are run. Several papers demonstrate that blocking concurrent algorithms can be faster [1, 14, 30, 51]. However, the experiments described in these papers are typically run in rarified environments in which all processes are dedicated to the task, often pinned to dedicated cores. They are also set up to have no page faults or other significant delays. In such environments, it is not surprising that algorithms using fine-grained (blocking) locks do well. Some have noted, however, that in environments with oversubscription (more processes than cores) blocking algorithms can suffer [14]. Our experiments verify this (e.g. see the right side of Figure 5d and 5g and the left side of Figure 5h). Of course, lock-based algorithms can also come to a grinding halt in environments where processes can be faulty.

In summary, for robustness in mixed environments, or for peace of mind in general, lock-free algorithms can have a significant advantage, but they come at the cost of more subtle and complicated designs, especially when used for more advanced data structures. Due to the tradeoffs, there is no universal agreement on whether lock-based or lock-free algorithms are better—some algorithms are lock-free [11, 20, 28, 29, 41, 51] and others use fine-grained locks [4, 9, 19, 30, 34, 37, 39, 42]. A third choice is to use transactional memory, but this has not yet shown itself to be competitive with either lock-free or lock-based approaches.

In this paper, we describe and study an approach that can get the best of both worlds—i.e., allow one to program with fine-grained locks while getting efficient lock-free behavior. We base our approach on a thirty-year-old idea of lock-free locks by Turek, Shasha and Prakash [49] and independently Barnes [3] (henceforth the TSP-B approach). However, we extend it significantly with several important new ideas to make it practical and more general. The high-level idea of the TSP-B approach is that when a thread takes a lock it leaves behind a descriptor that allows other threads that want the lock to help it complete its protected code and free its lock. The general idea of using descriptors for helping is now widely used in the implementation of specific lock-free applications, such as multiword-CAS [21, 26, 29, 50], other multiword operations [11], software transactional memory [23, 44, 46], and specialized data structures [6, 18, 20, 45, 52].

Despite the use of descriptors for helping in specific applications, we know of no general implementations of lock-free locks. Most of the papers cited above mention the TSP-B approach, often as motivation for their more specific approach, but describe it as impractical. The issue is that the TSP-B approach requires translating code in the lock into a form such that every read or write effectively requires saving the context of the process (program counter and local variables) so that others can help it run from that point. Such code can be very inefficient even when no helping occurs. Equally importantly, it makes the approach very difficult and clumsy to use without a special-purpose compiler. Their approach also constrains the code inside the locks to only allow race-free reads and writes to shared memory, which limits its applicability.

The key contribution of this paper is an approach to avoid the "context-saving" on each memory operation, making the approach practical, and additionally making it more general. In our approach the user can write standard code based on fine-grained locks, and using a pure library interface, get lock-free behavior. Beyond being efficient and offering a simple library-based interface, our approach generalizes the TSP-B approach by (1) allowing races in the locked code, (2) supporting memory allocation and freeing in the locked code, and (3) supporting try locks, which we demonstrate are much more efficient than the standard strict locks. The advantage of try lock is that it returns false if the lock is currently taken, giving the user the flexibility of either trying again or performing a different operation.

Our approach is based on a new technique to achieve idempotence. Intuitively, idempotent code is code that can be run multiple times but appears to have run once [5, 10, 16, 17]. Such code is important in the TSP-B approach since multiple helpers could run the same locked code when helping. TSP-B suggest particular approaches to idempotence (the approaches by TSP and B are quite similar) but failed to abstract out the notion of just needing idempotent code. Here, we abstract out the need of idempotence for lock-free locks and suggest a very different, as well as more efficient and general, approach to achieving idempotence. We also point out that to nest locks, we simply need the locking code itself to be idempotent, leading to locking code that is very simple.

In our approach to idempotence, instead of using "context saving" we maintain a shared log among processes running the same code. The log keeps track of all reads from shared mutable locations, as well as some other events, such as memory allocations. Whenever a copy of the thunk executes a loggable operation, it commits it to the log using a compare-and-swap (CAS). Whichever copy commits first wins, and all others take the value committed instead of their attempted commit. In this way, they all see the same committed values, e.g., the same reads, even though they are running in an arbitrary interleaved manner.

One key advantage of our approach is that the user can write concurrent algorithms based on fine-grained locks, and then either run it in a *lock-free mode* (with helping) or a *blocking mode* (no helping). The blocking mode can use a standard lock implementation without logging. The helping mode will log, at some additional cost, but guarantee lock-free behavior. Another key advantage over TSP-B is that our approach is based on try locks, instead of strict locks, which turns out to be important for the efficiency of optimistic use of fine-grained locks.

We have implemented our approach as a C++-based library called Flock. Based on the library we have implemented several data structures based on try-locks, including singly linked lists, doubly linked lists, binary trees, balanced blocked binary trees, (a,b)-trees , hash tables, and adaptive radix trees (ART). We compare performance of our versions in lock-free mode and blocking mode to the most efficient existing data structures we found, both lock-based and lock-free. The lock-based data structures generally perform slightly better under controlled environment with one process per processor, but perform significantly worse when oversubscribing with multiple processes per processor. Comparing running our algorithms in lock-free vs. blocking mode, the lock-free performance rarely has more than 10% overhead, and typically much less. However with oversubscription the lock-free mode greatly outperforms the blocking mode.

Our contributions include the following:

1. We present a new approach to achieving idempotence in general code, which relies on logging rather than context saving. For fine-grained locks it is significantly more practical and general than previous approaches.
2. We present a new approach to lock-free try-locks. They allow arbitrary nesting as long as there are no cycles in the precedence graph.
3. We develop a purely library-based general interface to support our ideas.
4. We compare several existing approaches with ours, both using locking and without locking.
5. We develop the first lock-free implementation of adaptive radix trees.

1.1 Example of Using Lock-Free Locks

To be concrete on how lock-free locks are used in our framework, we give an example of maintaining a concurrent sorted doubly-linked list supporting insert, delete, and find. The example uses optimistic fine-grained locks [34, 35]. Our C++ code using Flock is given in Algorithm 1. Each link holds a key and value, a next pointer, a lock, and a flag indicating whether the link has been removed. The `mutable_` wrapper around `next`, `prev`, and `removed` (lines 2–4) indicates that these are shared mutable values. They need to be read using a `load`, with a similar interface to a C++ `atomic`. Flock will log loads of mutable values when inside a lock. Since the key and value are immutable, they need not be put in `mutable_`.

Locks are attempted with the `try_lock` function. It takes a lock as an argument, as well as a *thunk* (a function with no arguments). In Flock, the thunk is simply a C++ lambda

```
1   struct link {
2     mutable_<link*> next;
3     mutable_<link*> prev;
4     mutable_<bool> removed;
5     Key k; Value v; lock lck;
6     link(Key k, Value v, link* next, link* prev)
7       : k(k), v(v), next(next), prev(prev), removed(false)
8       {};};

10  link* find_link(link* head, Key k) {
11    link* lnk = (head->next).load();
12    while (k > lnk->key) lnk = (lnk->next).load();
13    return lnk;}

15  std::optional<Value> find(link* head, Key k) {
16    link* lnk = find_link(head, k);
17    if (lnk->key == k) return lnk->value; // found
18    else return {}; }                     // not found

20  bool insert(link* head, Key k, Value v) {
21    while (true) {
22      link* next = find_link(head, k);
23      if (next->key == k) return false; // already there
24      link* prev = (next->prev).load();
25      if (prev->key < k &&
26          try_lock(prev->lck, [=] {
27            if (prev->removed.load() || // validate
28                (prev->next).load() != next)
29              return false;
30            link* newl = allocate<link>(k, v, next, prev);
31            prev->next = newl; // splice in
32            next->prev = newl;
33            return true;}))
34        return true;}}; // success

36  bool remove(link* head, Key k) {
37    while (true) {
38      link* lnk = find_link(head, k);
39      if (lnk->key != k) return false; // not found
40      link* prev = (lnk->prev).load();
41      if (try_lock(prev->lck, [=] {
42            return try_lock(lnk->lck, [=] {
43              if (prev->removed.load() || // validate
44                  (prev->next).load() != lnk)
45                return false;
46              link* next = (lnk->next).load();
47              lnk->removed = true;
48              prev->next = next;  // splice out
49              next->prev = prev;
50              retire<link>(lnk);
51              return true;});}))
52        return true;}} // success
```

Algorithm 1. Sorted doubly-linked lists using fine-grained optimistic locks with Flock. Flock code shown in red.

expression containing the code to be run when the lock is acquired, and returns a boolean indicating whether it was successful or not. If the lock is free, `try_lock` acquires the lock, runs the thunk, releases the lock, and returns the thunk's return value. Otherwise it returns false. The `try_lock` function forces locks to be properly nested. This is important for our lock-free locks since the thunk captures the code that might need to be helped by another `try_lock`. In Section 4 we describe a function that avoids pure nesting and supports, for example, hand-over-hand locking.

The `find_link` finds the first link with a key greater than or equal to the requested key. It requires no locks. The `find` just extracts the value from the link if the key matches.

The `remove` first finds the link `lnk` potentially containing the key. If it does not contain the key, then it returns `false` indicating the key was not in the list. Otherwise it tries to acquire a lock on the previous link (`prev`) and `lnk`. If either fails because they are already locked, the condition on line 41 will be false and the `while` loop will repeat. The conditions on lines 43 and 44 validate that the previous link has not been deleted, and `prev->next` still points to `lnk`. If either test fails then the `while` loop is repeated. If the tests pass, the code in the lock loads the next pointer from `lnk`, marks `lnk` as removed, splices it out of the doubly linked list, and retires its memory[1]. Note that a lock is not required on `next`. This is because a deletion of `next` or an insertion of an element before `next` would require a lock on `lnk` so it cannot happen concurrently. The `insert` is similar to `remove`.

This locking-based code for doubly-linked lists is much simpler than any lock-free versions we know of [2, 6, 25, 45, 47]. The difficulty in generating a lock-free version based on CAS is that lines 48–49 need to be applied atomically, as do lines 31–32. Our approach gives us a lock-free algorithm using the simple lock-based algorithm. As we show in our experiments, the lock-free version is almost as fast as the locking one without oversubscription, but much faster with oversubscription.

2 Model

We consider an asynchronous shared memory accessed by n processes. Processes can access the shared memory via the following atomic primitives: *read, write* and *compare-and-swap (CAS)*, defined in the standard way. We also assume a sysAlloc which returns a new memory location and a sysRetire which delays freeing the memory location until it is safe. Each process also has access to private memory, which it can make shared by modifying a shared location to point to it.

An *execution* is a sequence of *steps*, where each step specifies a primitive, its arguments, its return values, and the executing process. The steps taken by a process in an execution implement *operations*. An *event* is the *invocation* or *response* of an operation, which specify its arguments and return values, respectively, as well as its calling process. The first step of an operation in an execution is associated with its invocation and its last step is associated with its response. A *history* is a sequence of events, and can be derived from an execution E by including the invocations and responses

[1]Flock uses an epoch based memory manager. The `retire` puts the pointer asside and frees it when it is safe (after all concurrent operations finish).

of operations in the order their associated steps appear in E. An execution is *valid* if it is consistent with the semantics of the memory operations.

An object is a set of operations. Each operation is specified by a *sequential specification*, which defines its expected behavior in an execution in which the executing process's steps are not interleaved with the steps of any other process. An implementation of an object specifies code for processes to run for each of its operations. An implementation of an object O is *lock-free* if, in any infinite execution in which processes follow this implementation, infinitely many operations complete. This is equivalent to requiring a finite number of steps between responses of operations.

We say a memory location suffers from the *ABA problem* in some implementation if it is possible for the value written on that memory location to go back to what it was at some previous point in some execution of this implementation. We say an implementation suffers from the ABA problem if there is some memory location that suffers from the ABA problem in that implementation. An implementation is *ABA-free* if it does not suffer from the ABA problem.

3 Idempotence

To achieve lock-free critical sections, processes must be able to help each other. In particular, if some process holds a lock and crashes, others must be able to release the lock. Since it is possible that the crashed process has already begun its critical section, the other processes must complete its critical section for it before releasing the lock.

This leads to the need to have *idempotent* critical sections. Intuitively, a piece of code is idempotent if, when it is executed multiple times, it only appears to take effect once. Thus, if we have idempotent critical sections, processes can safely help execute someone else's critical section, without worrying about who else has also executed it. It is interesting to note that some code is naturally idempotent. In particular, a critical section that contains just one CAS instruction, which does not suffer from the ABA problem, is idempotent. After it is executed for the first time, subsequent executions of it would have their CAS fail, thus leaving the memory in the same state. Many hand-designed lock-free data structures achieve their lock freedom by allowing helping in such short, naturally idempotent sections.

However, in general, most code is not idempotent by default. For example, code incrementing a counter would yield different resulting counter values if it is executed several times. Thus, general lock-free constructions must be able to make general thunks idempotent. Several approaches in the literature have shown how to do so [3, 5, 7, 49]. In this section, we define idempotence formally and present a new construction that makes any piece of code idempotent.

3.1 Idempotence Definition

A *thunk* is a procedure with no arguments [32]. Note that any procedure with given arguments can be made a thunk by wrapping it in code that reads its arguments from memory. We assume that a thunk may have *thunk-local* memory which can only be accessed by processes executing the thunk.

We follow the definition of idempotence introduced in [5]. A *run* of a thunk T is the sequence of steps taken by *a single process* to execute or help execute T. The runs of a thunk by different processes can be interleaved and each run may take a different branch through the thunk depending on the memory state that it sees. A run is *finished* if it reached the end of T. An *instantiation* of thunk T in an execution E is a subsequence of E whose steps are the same as some run of T in E. Note that an instantiation of T is possibly composed of steps from different runs.

Definition 1 (Idempotence [5]). *A thunk T is idempotent if in any valid execution E consisting of runs of T interleaved with arbitrary other steps, there exists a valid instantiation E' of T that is a (possibly empty) subsequence of all operations from runs of T in E such that*

1. *if there is a finished run of T (response on T), then the last event of the first such finished run must be the end of E',*
2. *removing all of T's steps from E other than those in E' leaves a valid history.*

Intuitively, this definition allows a thunk T to be executed by several processes (in several runs of T), but other than one copy of each operation executed for T (a single instantiation), the rest have no effect. Furthermore, the instantiation of T that takes effect must end at the time that the first run of T by any process completed. This is to prevent the possibility that the effect or return value of T might 'wait' for some future event.

3.2 Our Approach to Idempotence

We now present a new approach to achieving idempotence in any code that is ABA-free. We note that it is easy to make code ABA-free by attaching a counter to any memory location that suffers from the ABA problem, and updating that counter every time the value is updated (our implementation does this). Rather than basing our idempotence construction on *context saving*, as were previous general idempotence constructions, we base our approach on using a shared *log*. We present pseudocode for the approach in Algorithm 2.

We store each thunk in an object, called the *descriptor*, that includes the thunk itself, and a *log*. The log keeps track of all values read, allocated or retired in any execution of the thunk. The log is *thunk-local*; any process executing this thunk uses the same descriptor object, so the log is shared by all processes that execute this thunk,[2] but no other process can access this log.

We implement five operations for idempotent code: *load*, *store*, *CAM* (a CAS that does not return any value), *allocate*, and *retire*. Any thunk can then be implemented using these operations. For ease of use, load, store and CAM are implemented in a struct called *mutable* that can wrap any type. Intuitively, we call it mutable since these are locations shared among processes that can be modified, yet still have to be idempotent. Any variable declared as mutable automatically uses our implemented operations rather than the corresponding primitives, keeping programmer effort to a minimum. We assume that CAMs and stores cannot race on the same location. Any non-mutable value, or any local variables/locations can be read and written as usual without using a *mutable*.

The idea is that each process keeps track of its current log (line 5) and how many items it has logged so far while running the corresponding thunk (its *position*, on line 6). Thus, when it starts executing a new thunk, it initializes its position to 0 and its local log to point to this thunk's log (lines 25 and 24). The process saves its previous log and position so that it can go back to the previous thunk when it finishes executing the new one. This is useful for executing nested thunks. Once a process has installed its new log and initialized its position, it can start running the thunk. Whenever it executes a new loggable instruction (load, allocate or retire), it uses the shared log of the thunk to record the return value of this instruction and to see whether others have already logged it.

To do so, it makes use of a key helper function called *commitValue* (line 34). This function is what the process uses to log its thunk's loggable instructions. This function takes in a value to be logged; intuitively, this is the intended return value of the current instruction. The process uses its current position to index into its thunk's log. It tries to commit its value by using a CAS on `log[position]`, with old value `empty`, and new value equal to the value it would like to log. All log entries are initialized to `empty` and we assume that no process attempts to write `empty` into a mutable variable. The process then checks what value is written in `log[position]`, and returns this value, as well as a boolean indicating whether or not its CAS succeeded (i.e. whether it was the first to execute this instruction on this thunk). When the process does not currently have a log (i.e. is not currently executing a thunk), the `commitValue` function simply returns the input value and the success flag set to true (line 33). This case happens when the instruction is executed outside of all locks. For example, no logging is needed for the loads on line 12 in Algorithm 1 since they are not in a lock, but the load on line 43 is logged in the descriptor for its surrounding lock.

[2]Note that this differs from distributed logs used in, for example, optimistic transactional memory [35], where each process has its own log. It also differs from logs used to commit successful transactions.

To load a value from a given mutable field, a process simply reads this field, and then tries to commit its value to the log by calling `commitValue`. The return value of the load (line 35) is the value returned by the `commitValue` call. In this way, the process returns the same value from its load as any other process executing this load for this thunk.

To store a value in a given mutable field, the process first executes a load as described above, thereby logging the value present before the store occurred, or discovering what that value was (if this store was already executed by a different process). The process then executes a CAS with old value equal to the value returned from the load. Recall that we assume that shared memory locations are ABA free, and therefore this ensures that all CAS attempts but the first will fail. The CAM operation works similarly to the store, but with an additional check to make sure the value returned by the load matches the expected value. It only executes a CAS if this is the case. By performing a load before the CAS, we guarantee that the expected value was stored in the memory location at some point. Combined with the ABA-free assumption, this prevents a potentially dangerous scenario where the expected value is written into the memory location after the CAS, causing future executions of the CAM to no longer be idempotent. It is important that the CAM does not return the return value of its CAS, since this value could be different for different processes that execute it, and could therefore violate idempotence (externalize a different result).

We also provide allocate and retire operations for idempotence. The idea is again to use the thunk log to commit values. To allocate a new object, the process allocates this object using the system-provided allocation mechanism, and then uses `commitValue` to install this new object in the log. If it is the first to do so, then the allocation is done, and this new object is returned. Otherwise, the process destroys its newly allocated object, and instead returns the object that was already installed in the log.

To retire an object, the processes use the log to compete for 'ownership' of this object. The first process to commit a boolean retirement flag on the log is responsible for retiring this object. All other processes simply skip retiring it if they discover, by trying to commit a flag to the log, that some other process already owns this object. In this way, each object is retired at most once. Standard garbage collection techniques can then be used to collect retired objects when it is safe to do so.

The commitValue can also be used directly by the user to commit the result of any non-deterministic instruction. For example if there is an instruction that generates a bit based on random noise in the processor, this needs to be committed so all instances of the thunk agree on it.

```
type Log = entry[logSize];
type Thunk = function with no arguments returning bool

private process local:
  Log* log; // the current log for a process
  int position; // the current position in the log

struct descriptor:
  Log* log;
  Thunk thunk;
  boolean done;

descriptor* createDescriptor(Thunk f):
  Log* log = allocate<Log>();
  return allocate<descriptor>(log, f, false);

void retireDescriptor(descriptor* T):
  retire<Log>(D->log);
  retire<descriptor>(D);

bool run(descriptor* D):
  V* old_log = log; // store existing log and position
  int old_pos = position
  log = D->log; // install D's log
  position = 0;
  bool returnVal = D->thunk(); // run thunk
  log = old_log; // reinstall previous log and position
  position = old_pos
  D->done.write(true);
  return returnVal;

<V, bool> commitValue(V val):
  if (log == null): return <val, true>;
  bool isFirst = CAS(log[position], empty, val);
  V returnVal = read(log[position]);
  position++;
  return <returnVal, isFirst>;

struct mutable<V>:
  V val;
  V load ():
    V v = read(val);
    return commitValue(v).first;
  void store(V newV):
    V oldV = load();
    CAS(val, oldV, newV);
  void CAM(V oldV, V newV):
    V check = load();
    if (check != oldV): return;
    CAS(val, oldV, newV);

V* allocate<V>(args):
  V* newV = sysAllocate(args) //use system allocator;
  <obj, isFirst> = commitValue(newV);
  if not isFirst: sysFree(newV);
  return obj;

void retire<V>(V* obj)
  <_, isFirst> = commitValue(1);
  if isFirst: sysRetire(obj);
```

Algorithm 2. Idempotent primitives. The **entry**s of the log are assumed to hold any type that fits in a word (or two if using double width CAS). In the code, the log is of fixed sized, but could grow by adding blocks as needed (see Section 6 for details).

We now show that our idempotence construction is correct; that is, any thunk that replaces all its variables with mutable ones according to Algorithm 2 becomes idempotent.

Theorem 3.1. *Replacing each variable in a thunk T with a mutable variable and allocating and retiring all objects in T with the provided allocate and retire operations yields an idempotent version of T.*

Proof. (Outline.) A complete proof which lines up with our definition of idempotence is given in the full version of our paper [?]. Here we give a brief outline. The idea is that all processes running the same thunk (descriptor) will stay synchronized in the sense they will have the same state at the same point of their execution. Whichever gets to a loggable event first will log it, and all others will see it is already logged and use the same value. In this way, they all see the same values, and stay synchronized. It also means their position in the log will be synchronized. Memory allocation and retiring is safe since only the first will keep its allocated value and only the first will retire the value. For stores and CAMs, only the first such operation will succeed and all others will fail, because of our ABA-free assumption. Therefore only the first will be visible. The full proof also shows the operations are linearizable (assuming stores and CAMs on the same location are not concurrent). □

As mentioned, most previous approaches to idempotence have been based on *context saving* [3, 5, 7, 8, 49]. This involves storing out a program counter and current state of all local variables at important events (e.g. shared memory operations), and possibly loading and installing a new context if already stored. Our approach never needs to store a program counter or local state since the processes are running "synchronously" and have the same local state. For large thunks, and frequent helping, however, our method potentially does have an additional cost. In particular, we always start helping from the beginning of a thunk while the other methods will start at the point of the last context saved by any process. Our method is therefore particularly well suited for short thunks, which is the intended use with fine-grained locks, and possibly not as well suited for long running thunks.

4 Lockless Locks

We now describe how we implement a tryLock. It is important that tryLocks can be nested, which means the locking mechanism itself must be idempotent or otherwise safe to use when there are multiple threads helping to acquire the lock. In particular, consider an operation O_1 that takes an outer lock L_a and inside the lock takes an inner lock L_b. If another operation O_2 encounters L_a locked, it will help O_1 execute its critical code. This means it will help O_1 acquire L_b and, if successful, run the code of O_1 in that lock.

Based on our technique for idempotence, it turns out to be quite simple to implement the locking mechanism. In particular, we simply need to ensure that the code for the locking

```
1  struct lockDescr :
2     descriptor* d;
3     bool isLocked;

5  type Lock = mutable<lockDescr>;

7  bool tryLock(Lock* lock, Thunk f):
8    bool result = false;
9    lockDescr currentDescr = lock->load();
10   if (not currentDescr.isLocked) :
11     descriptor* myDescr = createDescriptor(f);
12     lockDescr myLockedDescr = {myDescr.d, true};
13     lock->CAM(currentDescr, myLockedDescr);
14     currentDescr = lock->load();
15     if (myLockedDescr.d->done or
16          myLockedDescr == currentDescr) :
17       result = run(myDescr.d); //run self
18       lock->CAM(myLockedDescr,
19                 lockDescr(myDescr.d, false)); //unlock
20     retireDescriptor(myDescr.d);
21   if (currentDescr.isLocked)
22     run(currentDescr.d); //help
23     lock->CAM(currentDescr,
24               lockDescr(currentDescr.d, false)); //unlock
25   return result;

27 void unlock(Lock* lock):
28   lockDescr currDescr = lock->load();
29   lock->CAM(currDescr, lockDescr(currDescr.d, false));
```

Algorithm 3. Idempotent TryLock

mechanism is itself idempotent, so that helping it is safe. Our code is given in Algorithm 3. A lock descriptor (`lockDescr`) is represented as a pair of a pointer to a descriptor and a boolean indicating whether it is currently locked or not. It is easy to put these into a single word by stealing a bit from the pointer. A `Lock` is then a `mutable` lock descriptor.

An attempt at acquiring the lock starts by reading the lock and checking if it is currently locked. If not locked, the algorithm creates a descriptor for the thunk f (line 11) and tags it to mark that it is locked (line 12). It then attempts to install the descriptor on the lock using a CAM (we do not have a CAS for mutables). Since the CAM does not return whether it succeeds, the algorithm needs to read the lock again (line 16) to check if successfully acquired. If acquired, it runs the code by calling `run` on `myDescr`, then releases the lock (line 19). Whether the CAM was successful or not, `myDescr` needs to be retired (line 20). If on line 10 the lock is already locked, then the algorithm helps the tryLock that has it acquired (line 22), and releases the lock (line 24). Finally the `result` is returned, which will only be true if the lock was successfully acquired and the thunk f returns true.

We now argue correctness. We say a tryLock is correct if it either fails, in which case none of its the critical code (thunk f) is run and it returns false; or it succeeds, in which case all its critical code is run and the tryLock returns its value. If successful, no critical code on the same lock can run concurrently. By this definition, the tryLock could always fail, but this would not satisfy progress bounds, and in particular for us, our lock-free bounds. We say a successful tryLock *enters* on the step the lock is changed to point to its descriptor and *exits* on the step when the lock is changed from locked with its descriptor to unlocked.

Theorem 4.1. *The tryLock in Algorithm 3 is correct as long as* `run(descriptor)` *runs the user code in the thunk f idempotently, and the operations on a Lock (*`load`*,* `CAM` *and* `store`*) and on descriptors (*`createDescriptor` *and* `retireDescriptor`*) are idempotent.*

Proof. (Outline). The code in a thunk consists of the user level code and possibly the code of a nested tryLock. Together this is idempotent by assumption. In the algorithm, a descriptor is run if and only if the tryLock enters and the lock is set. The descriptor could be run both in line 17, and in line 22 by another process. Some process will finish the thunk first (either the primary process or a helper). Since the thunk is idempotent, any processes working on the same descriptor after that point will have no effect. The lock is only released after the thunk is first finished so the code can only have an effect between when the successful tryLock enters and exits. Since there is a unique descriptor on the lock during this time, no other thunk on the same lock can appear to run concurrently (there could be leftover thunks from earlier successful attempts on the lock, but they will have no effects). If either the check on line 10 or on line 16 fail then the tryLock fails and returns false. □

The theorem does not depend on a particular implementation of idempotence, but works with ours since ours satisfies the specified conditions.

We now show that tryLocks are lock-free. We allow for arbitrary nesting of locks, but as with standard locks we require that there are no cycles in the precedence graph on the locks. In particular, consider a precedence graph in which locks are vertices and we place an edge from lock L_a to lock L_b if a tryLock on L_b is directly called while running the code for a tryLock on L_a. The edge exists from when tryLock (L_b) is invoked until it responds. We say a concurrent algorithm (or data structure) is *lock-cycle free* if a cycle in the precedence graph cannot occur at any time in any history. As is standard this can be ensured by acquiring locks in some ordering. If the locks are known ahead of time, they can be sorted. If the locks are in a data structure with an ordering (e.g. directed tree, DAG, etc) they can be taken in an ordering consistent with the data structure. In a rooted tree, always locking a parent before its children is sufficient.

We also need to bound the time of user code in a lock, otherwise helpers could never complete helping. We defined *step count* for a tryLock as the number of user steps taken by its critical region. We count all functions in the idempotent interface as unit cost plus any user code inside of them—in particular `sysAllocate` and `sysRetire` count toward user code. We count a nested `try_lock` as unit cost plus the step count of the code in its critical region.

We say a tryLock is a *leaf* if successfully completes. Note that a leaf tryLock succeeding is significant since it means all its ancestors must have succeeded, otherwise it would not even have been invoked. We therefore measure progress by the completion of leaf tryLocks. A tryLock is a *root* if it is not called from within a critical region. Every tryLock has an *owner*. The owner of a root tryLock is the process than runs it, and every other tryLock inherits the owner from the tryLock that calls it (not necessarily the process that calls it).

Theorem 4.2. *Consider an algorithm using tryLocks, which is lock-cycle free, and for which the maximum step count for any tryLock is bounded. On p processes, a tryLock will run in bounded steps and every at most p root trylock attempts will successfully complete a distinct leaf tryLock.*

Proof. (Outline.) A complete proof is given in the full version of our paper [**?**]. Here we give an outline. First we define a notion of helping chain, which is a chain of recursive helping a process goes through when it encounters a lock that is occupied. We show that each tryLock on the chain has a unique owner, so therefore a chain can only be $p - 1$ long. Since we assume the time for each tryLock is bounded, this bounds the time for helping. Furthermore we show that the tail of the chain is a leaf tryLock, such that helping will always be responsible for completing a leaf. Finally we argue that at most p root tryLocks can follow the same chain and therefore end up at the same leaf. ◻

We note that this theorem indicates that tryLocks are lock-free (under the required conditions) in that a leaf must complete in a finite number of steps. It does not, however, imply wait-freedom since a particular process could continuously fail to acquire a lock. It also does not, by itself, guarantee an algorithm using tryLocks is lock-free even if the algorithm satisfies our conditions. In an opportunistic algorithm based on fine-grained locks, for example, we might need to retry not because a lock failed to be acquired, but instead because some consistency check failed (e.g. the test on line 43 of Algorithm 1). In all the algorithms we consider, however, a consistency check can only fail if the algorithm has made progress. In the `remove` from Algorithm 1 the consistency check can only fail if in between the `find_location` and when the lock on `cur` is acquired, either (1) `cur` is deleted or (2) it is updated to point to a new `next`. In either case, the algorithm has made progress by completing an operation. A similar argument can be made for the `insert`. Therefore the ordered list algorithm based on our tryLocks is lock-free, as are the other algorithms we consider.

It can be useful to release a lock early before the scope of the thunk associated with the acquired lock completes. We supply a `unlock` for this purpose. It takes a lock that is currently acquired by the thread and unlocks it. Its behavior is undefined if the thread has not acquired the lock. As mentioned in the introduction, this can be used for hand-over-hand locking (also called lock-coupling) [4].

The code for tryLock can be modified to support a strictLock that always acquires the lock before returning, by first creating the descriptor, and then putting the attempt to acquire a lock into a while loop. We have implemented such a strictLock and compare it to the tryLock in Section 8.

5 Related Work

As mentioned, the idea of lock-free locks was introduced by Turek et al [49] and Barnes [3]. The idea of helping dates back earlier, at least to Herlihy's work on wait-free simulations [31]. Many wait-free and lock-free algorithms achieve their progress guarantees by allowing processes to safely *help* each other complete their operations, although in quite specific ways instead of using a general mechanism. Help used for wait-free progress was formally studied by Censor et al. [12].

The idea of *idempotence* has been used in the literature a variety of contexts [7, 8, 8, 10, 16, 17, 33, 38]. Kruijf, Sankaralingam and Jha [17] give a nice overview although only up to 2012. More recent work has focused on using idempotence for fault tolerance (e.g., [7, 8, 38]). All these approaches rely on some form of "context saving". Idempotence has also been considered and characterized in the literature under different names. Timnat and Petrank [48] define a similar notion known as *parallelizable* code, which intuitively allows several processes to execute it without changing its effects.

In recent work Ben-David and Blelloch in [5] use a randomized implementation of lock-free locks to show that when point contention on locks in constant, then operations can be completed in constant expected time. We use their definition of idempotence in this paper. However, their focus is on theoretical efficiency and fairness guarantees of acquiring the locks, whereas in this paper we focus on the practicality of the approach. As with previous approaches to idempotence their approach relies on context saving.

Approaches for achieving idempotence and lock-freedom sit on a spectrum of generality. The focus of this paper is to improve the practicality of the far side of the spectrum; fully general idempotence/lock-free constructions. However, many other approaches exist, which are less general but can be more efficient for their specific applications. For example, on the other end of the spectrum are hand-designed lock-free data structures. These data structures are often designed to be able to have 'critical sections' that contain just one CAS instruction, and can therefore be executed atomically in hardware with no locks. For example, Michael and Scott's queue [40] allows new nodes to be enqueued by swinging a single pointer. Idempotent help is given by later updating the tail pointer. Similar algorithms, like Harris's linked-list [28] and Natarajan and Mital's BST [41], make use of descriptors to allow others to help, but these descriptors are optimized to simply be flags. These approaches yield very fast lock-free data structures, but are difficult to generalize.

A middle-ground between generality and efficiency is found with approaches that implement useful primitives for lock-freedom. For example, Brown et al [11] introduce the LLX/SCX primitive, which allows atomically checking that several locations have not changed their values, 'freezing' some of them, and modifying one of them. This primitive can be seen as a lock with a restricted critical section. Another example of such a primitive is multi-word CAS, which allows several memory locations to be CASed atomically [21, 26, 29].

Some work aims at achieving practical lock-free locks but only partially solve the problem. Rajwar and Goodman describe a hardware-based technique that are lock-free under an assumption that processes do not fail or stall during certain critical regions [43]. We assume a process can fail or stall at any instruction. Gidenstam and Papatriantafilou [24] look at how to make the handoff of locks lock-free (i.e., waking up threads suspended on a lock in a lock-free manner), but a thread blocked during a lock will still delay any waiting threads indefinitely.

6 The Flock Library

We have implemented a C++ library, Flock, based on our lock-free locks approach. It supports a `mutable_` wrapper to use on any shared values that can be mutated inside a lock, a `lock` type and a `try_lock`. The `mutable_` wrapper has a similar interface to the C++ `atomic` wrapper. In particular, it supports `load`, `store` and `cam`. The assignment operator (=) is overloaded to store. Flock also supports `allocate` and `retire` which are integrated with its epoch-based collector. An example of how to use Flock is given in Algorithm 1. The library is available at https://github.com/cmuparlay/flock.

Here we discuss several specifics about the implementation, including some optimizations.

Epoch-based collection. Flock uses an epoch-based memory manager [22, 27]. In such a memory manager, each operation runs in an epoch, each of which is associated with an integer that increases over time. Managing memory with epochs requires some additions to the the implementation of idempotent code. In particular, when a thread helps another thread, it is taking on the responsibility of that other thread. It therefore needs to also take on its epoch number. To implement this, when Flock has to help inside of a `try_lock`, it changes its epoch to be the minimum of its epoch and the epoch of the thunk it is helping. When it is finished helping, it restores its epoch to what it was before helping. The descriptors are also allocated and retired with the same epoch-based collector, with one optimization. In particular if a descriptor is never helped, which is the common case, then it can be reused immediately instead of being retired. To implement this, we keep a flag on the descriptors which is set when helping. This requires some careful synchronization.

ABA. Although the idempotent implementation in Algorithm 2 requires that mutables are ABA free, a `mutable_` in Flock does not have this requirement. To allow for this, Flock keeps tags on mutable locations. A simple implementation is to use a 64-bit counter, and increment the counter on each update. Assuming mutable values can be up to 64-bits, this can be implemented with double-word (128-bit) loads and CASes. Unfortunately double-word loads are particularly expensive on current machines. Flock has two optimizations to avoid them, one which supports 64-bit values, and one for 48-bit values, which is sufficient for a pointer.

The first optimization still uses a 64-bit counter on every mutable, but avoids any double-word loads. A key observation is that a `load` only needs to log the value, and therefore only needs to read this value. Another observation is that a `store` (or `cam`) does not need to read the counter and value atomically. Instead, it can first read the counter and then the value, followed by a double-word CAS to the mutable. This is safe since the value can only change if the counter changes.

The second optimization avoids the extra 64-bit counter on each mutable location and any double word operations. Instead it uses a safe lock-free approach that only requires a 16-bit tag. The full description is beyond the scope of this paper, but roughly it uses an announcement array to ensure that wrapping around is safe—i.e., it never uses a tag that is announced. All the experiments in Section 8 use this version since the mutables are no larger than a pointer.

Constants and Update-once Locations. Shared, constant locations do not need to be wrapped in a `mutable_` and can just be read directly. A constant location is one that is written once and only read after it is written. The write could happen during construction of the object that contains it or after. For example the key and value in the list link in Algorithm 1 are constants. Flock also supports update-once locations. These are locations that have an initial value, and are updated at most once. Reads can happen before or after the update. The `removed` flag in a link in Algorithm 1, for example, is updated once. Update-once variables are ABA free and therefore do not need a tag. Furthermore, the `store` can be implemented with a simple write instead of a `load` and then a `CAS`. This is because only the first such write will have an effect.

Arbitrary Length Logs. In general it cannot be determined ahead of time how long a log will be. Flock therefore implements logs that can dynamically increase in size. In the implementation, a log has a fixed block size (7 by default). If it runs out, another block is allocated. To do this idempotently, the first thread that runs out allocates the block and attempts to CAS it into a next-block pointer. If it fails, it frees its block and takes on the block that succeeded.

Avoiding CASes. We found that one of the most expensive aspects of helping is contention due to CASes on both the log and mutable locations. This is especially true under high contention when there is a lot of helping. To significantly reduce this contention we use a compare-and-compare-and-swap. In particular, before doing a CAS, the location is read and compared against the expected value, and if not equal the CAS can be avoided. When helping under high contention it is often not equal (someone else already executed the CAS) so many of the CASes are avoided. This rather simple change made a significant improvement in performance under high contention—sometimes a factor of two or more.

Capturing by Value. In the code in Algorithm 1, one might notice the "`[=]`" in the definition of the lambda's. This indicates that all free variables in the lambda defined outside of it are captured by value, as opposed to by reference—i.e., they are copied into the thunk. This is important since the lambda might outlive its context, and any surrounding stack allocated values could be destructed while being helped. Indeed if the `[=]` is replaced by `[&]` (by reference), Algorithm 1 would be incorrect—for example, the variable `prev` on line 40 could be reused while the lambda is being helped.

7 Data Structures

We have implemented several concurrent data structures using Flock. These data structures include the doubly linked list described in Section 1.1 (`dlist`), a singly-linked list (`lazylist`), an adaptive radix tree [36, 37] (`arttree`) which is a state-of-the-art index data structure used in the database community, a separate chaining hashtable (`hashtable`), a leaf-oriented unbalanced BST (`leaftree`), and a leaf-oriented balanced BST (`leaftreap`) with an optimization that stores a batch of key-value pairs (up to 2 cachelines worth) in each leaf to minimize height. To support concurrent accesses, the data structures use fine-grained, optimistic locking, as in [9, 30, 34, 37]. This approach involves (1) traversing the data structure without any locks, (2) locking a neighborhood around the nodes you wish to modify, (3) checking for consistency, and (4) performing the desired modifications. If the consistency check fails, locks are released and the operation restarts. Read-only operations do not take any locks. We use EBR [23] for safe memory reclamation.

We implement a tryLock and a strictLock version of each data structure. Both tryLock and strictLock can either be lock-free (with helping) or blocking (using test-and-test-and-set locks), and with our library, this choice can be made at runtime.

To the best of our knowledge, this results in the first lock-free implementation of an adaptive radix tree. In many workloads, our lock-free `arttree` significantly outperforms the other lock-free ordered set data structures that we ran. We plan to make our implementations publicly available.

8 Experimental Evaluation

Our experimental evaluation has two main goals: first, to compare the performance of lock-free locks with blocking locks and second, to compare data structures written with lock-free locks with state-of-the-art alternatives.

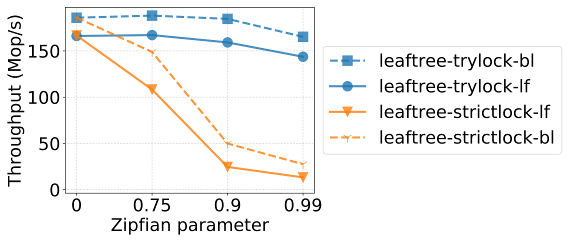
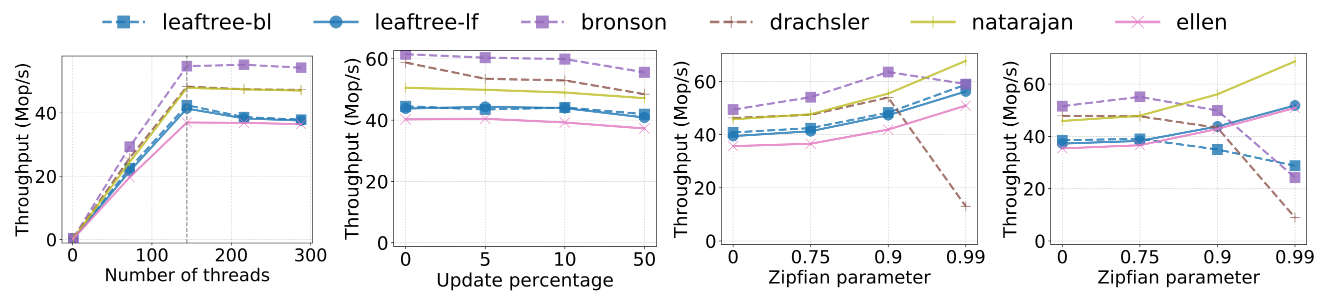


Figure 4. Comparing try lock with strict lock on workload with 100K keys, $\alpha = 0.75$, and 50% updates. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.

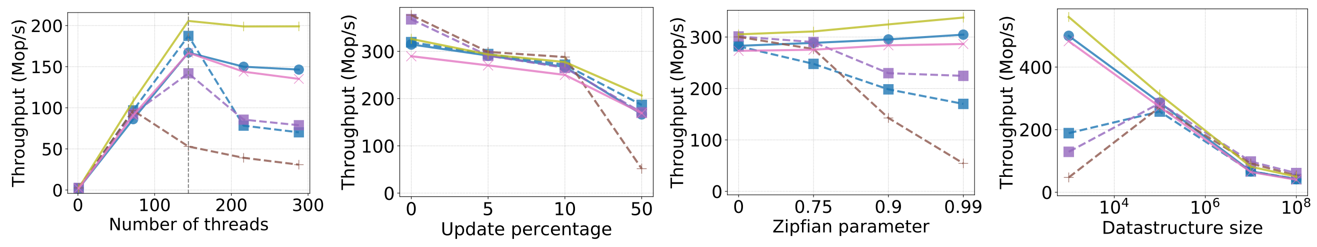
Setup. Our experiments ran on a 72-core Dell R930 with 4x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. The machine's interconnection layout is fully connected so all four sockets are equidistant from each other. We interleaved memory across sockets using `numactl -i all`. The machine runs Ubuntu 16.04.6 LTS. We compiled using g++ 9.2.1 with `-O3`. Jemalloc was used for scalable memory allocation. In our linked lists experiments, we padded nodes to cache line boundaries to prevent false sharing, but for the trees and hashtables, nodes were not padded to maximize the amount that fit in cache.

Workloads. We experiment with set data structures supporting `insert`, `delete` and `lookup` on 8-byte key-value pairs. Our experiments follow a similar methodology to previous papers [1, 15]. We first pick a key range $[1, r]$ and prefill the data structure with half the keys in the range. Then each thread performs a mix of `lookup` and update operations, where update operations are evenly split between `insert`s and `delete`s, keeping the data structure size stable throughout the run. Each experiment is run for 3 seconds (sufficient for reaching a stable state) and an average of 3 runs is reported (variance between runs was small). All keys are randomly chosen from the range $[1, r]$ according to a zipfian distribution parameterized by α. Zipfian with $\alpha = 0$ is identical to the uniform distribution and higher α skews accesses towards certain "hot" keys, which is more representative of real-world workloads. The zipfian distribution is also used in the YCSB benchmark suite, which mimics OLTP index workloads [13]. We mostly run with 5% and 50% updates, following YCSB Workloads B and A, respectively.

Our experiments vary four parameters: data structure size, update rate, α, and number of threads. We shows graphs along each of the dimensions, fixing the other three. Since `arttree` is a trie data structure, it benefits heavily from densely packed keys, so we sparsify the key range by hashing each key from $[1, r]$ to a 64-bit integer. This doesn't affect the other data structures since they either are purely comparison based or hash the keys themselves.

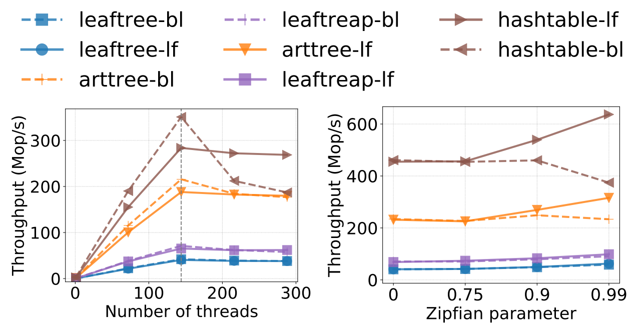


(a) 100M keys, 50% up., $\alpha = 0.75$ **(b)** 100M keys, 144 th., $\alpha = 0.75$ **(c)** 100M keys, 144 th., 50% up. **(d)** 100M keys, 216 th., 50% up.



(e) 100K keys, 50% up., $\alpha = 0.75$ **(f)** 100K keys, 144 th., $\alpha = 0.75$ **(g)** 100K keys, 216 th., 5% up. **(h)** 216 th., 5% up., $\alpha = 0.75$

Figure 5. Throughput of binary trees under a variety of workloads are shown. Dotted lines are used for blocking data structures and solid lines for lock-free ones. Subcaptions abbreviate 'threads' to 'th' and 'updates' to 'up'.



(a) 100M keys, 50% up., $\alpha = 0.75$ **(b)** 100M keys, 216 th., 5% up.

Figure 6. Throughput of various concurrent set data structures.

Try vs strict lock. In data structures that employ optimistic locking, tryLock is often preferable to strictLock. This is because optimistic locking requires checking for consistency after taking the necessary locks. So if a process p_1 tries to acquire a lock that is held by a another process p_2, it is better for p_1 to restart its operation instead of waiting to acquire the lock because it will likely fail its consistency check due to modifications by p_2. We see this happen in the `leaftree` in Figure 4. The higher α is, the more contention there is on the locks, and the more beneficial tryLock becomes. This holds for both blocking locks and lock-free locks. In the rest of this section, we only use tryLock in the data structures we implement. Note that blocking locks are faster than lock-free locks in this experiment. This is discussed in more detail below.

Binary trees. Figure 5 shows the throughput of concurrent trees under a wide range of workloads. We compare our tree implementations with state-of-the-art lock-based (Bronson [9], Drachsler [19]) and lock-free (Ellen [20] and Natarajan [41]) binary search trees. These implementations were obtained from the following benchmarking suite [1]. Bronson is the only balanced tree among these implementations. In general, our `leaftree` using lock-free locks is competitive with the state-of-the-art lock-free trees, which are implemented directly from CAS.

Figures 5a- 5d consider the case where the tree does not fit in cache. In this case, performance is dominated by cache misses incurred during the traversal phase. Figure 5b shows that the cost of modifying the tree is small compared to these cache misses. All trees scale well, up until oversubscription (Figure 5a). Bronson is generally the fastest when tree size is large because it is better balanced compared to the other trees (which are only balanced in expectation due to random inserts), resulting in shorter traversals and less cache misses. As the zipfian parameter α increases, most trees tend to speed up because higher α means more locality and less cache misses (Figure 5c). However, large α also means more contention. In the case of Bronson and Drachsler, which both use strict-locks, this extra contention out-weighs the benefits of locality. Results for small tree sizes that do fit in cache are shown in Figures 5e- 5g.

Lock-free vs blocking. Next, we compare the performance of lock-free data structures with blocking ones, with particular emphasis on `leaftree-lf` and `leaftree-bl`. The overhead of lock-free locks come from two main sources (1) allocating and initializing a new descriptor every time a lock is acquired, and (2) committing values to the log during critical sections. A successful `insert` commits about 5 entries to the log. If lock contention is rare, the cost of the two steps is very small, as seen in read-mostly workloads (Figure 5f) and in big tree sizes (Figure 5a). However, if two processes contend on the same lock and one helps the other, then both will access the same log, invalidating each other's cache lines and slowing down the logging step. As contention increases in Figures 5c and 5f, this overhead increases up to about 10%. The worst case for lock-free locks is shown in Figure 4 where it is half as fast as blocking locks when $\alpha = .99$ (most popular key accounts for 8% of accesses).

Where lock-free algorithms shine is in oversubscribed cases (e.g. 288 threads) with high contention. This is because a thread may get descheduled while it is partway through an update, and in a lock-free algorithm, if another thread wants to update the same location, it can simply help complete the inactive thread's update and then proceed with its own. However, in a blocking data structure, the new thread will have to either wait for the inactive thread to be scheduled again and release its lock or yield and context switch, both of which are expensive. This effect can be seen in Figures 5d and 5g where the three lock-free trees outperform the three blocking trees when contention (α) is high. In particular, `leaftree-lf` outperforms `leaftreap-bl` by up to 50% at $\alpha = .99$. A similar effect can be seen in Figure 5h where high contention is caused by small tree size.

Other set datatypes. In Figure 6, `arttree`, `leaftreap`, and `hashtable`, generally follow the same pattern as `leaftree`. That is, lock-free versions outperform their blocking counterparts in oversubscribed, high contention scenarios (Figure 6b), and the overhead of using lock-free locks in non-oversubscribed scenarios is small (Figure 6a). One exception is the `leaftreap` in Figure 6b in the case where $\alpha = .99$. In this case, contention caused by helping actually outweighs the benefits which causes `leaftreap-bl` to outperform `leaftreap-lf`. Experimental results for singly and doubly-linked lists written using our library can be found in the supplementary materials.

References

[1] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In Haryadi S. Gunawi and Benjamin Reed, editors, *USENIX Annual Technical Conference*, 2018.

[2] Hagit Attiya and Eshcar Hillel. Built-in coloring for highly-concurrent doubly-linked lists. *Theory of Computing Systems*, 52(4):729–762, 2013.

[3] Greg Barnes. A method for implementing lock-free shared-data structures. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 261–270, 1993.

[4] R. Bayer and M. Schkolnick. *Concurrency of Operations on B-Trees*, page 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[5] Naama Ben-David and Guy E. Blelloch. Fast and fair lock-free locks, 2021.

[6] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. Space and Time Bounded Multiversion Garbage Collection. In *International Symposium on Distributed Computing (DISC)*, pages 12:1–12:20, 2021.

[7] Naama Ben-David, Guy E Blelloch, Michal Friedman, and Yuanhao Wei. Delay-free concurrency on faulty persistent memory. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2019.

[8] Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 247–258, 2018.

[9] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2010.

[10] Jeremy Brown, J. P. Grossman, and Tom Knight. A lightweight idempotent messaging protocol for faulty networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2002.

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[12] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 241–250, 2015.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News*, 43(1):631–644, 2015.

[16] Marc de Kruijf and Karthikeyan Sankaralingam. Idempotent code generation: Implementation, analysis, and evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[17] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[18] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd international symposium on Memory management*, pages 163–174, 2002.

[19] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014.

[20] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2010.

[21] Steven Feldman, Pierre LaBorde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43(4):572–596, 2015.

[22] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[23] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5–es, 2007.

[24] Anders Gidenstam and Marina Papatriantafilou. Lfthreads: A lock-free thread library. In Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal, editors, *Conf. on Principles of Distributed Systems (OPODIS)*, 2007.

[25] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, pages 260–269, 2002.

[26] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. Efficient multi-word compare and swap. *International Symposium on Distributed Computing (DISC)*, 2020.

[27] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[28] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing (DISC)*, pages 300–314. Springer, 2001.

[29] Timothy L Harris, Keir Fraser, and Ian A Pratt. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing (DISC)*, pages 265–279, 2002.

[30] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Conf. on Principles of Distributed Systems (OPODIS)*, 2006.

[31] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, jan 1991.

[32] Peter Zilahy Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[33] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Exploiting reference idempotency to reduce speculative storage overflow. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(5):942–965, sep 2006.

[34] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3), 1980.

[35] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[36] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *IEEE International Conference on Data Engineering (ICDE)*, 2013.

[37] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The art of practical synchronization. In *Proc. International Workshop on Data Management on New Hardware*, 2016.

[38] Brandon Lucia and Benjamin Ransford. A simpler, safer programming and execution model for intermittent systems. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2015.

[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[40] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[41] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2014.

[42] William Pugh. Concurrent maintenance of skip lists. Technical Report TR-CS-2222, Dept. of Computer Science, University of Maryland, College Park, 1989.

[43] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In Kourosh Gharachorloo and David A. Wood, editors, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17. ACM Press, 2002.

[44] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 151–163. IEEE, 2019.

[45] Niloufar Shafiei. Non-blocking doubly-linked lists with good amortized complexity. *arXiv preprint arXiv:1408.1935*, 2014.

[46] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[47] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

[48] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 49(8):357–368, 2014.

[49] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Principles of Database Systems (PODS)*, pages 212–222, 1992.

[50] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 461–472. IEEE, 2018.

[51] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. Building a bw-tree takes more than just buzz words. pages 473–488, 05 2018.

[52] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. *ACM Transactions on Parallel Computing (TOPC)*, 8(2):1–38, 2021.

```
void insert(link* head, K k, V v) {
  while (true) {
    auto [cur, nxt] = find_location(head, k);
    if (nxt->k == k) return;
    if (try_lock(cur->lck, [=] () {
        if (cur->removed.load() ||
          (cur->next).load() != nxt) return false;
        link* new_link = link_pool.allocate(k, v, nxt);
        cur->next = new_link;
        return true;}))
      return;}}

std::optional<V> find(link* head, K k) {
  link* nxt = (head->next).load();
  while (k > nxt->key) nxt = (nxt->next).load();
  if (nxt->key == k) return nxt->value;
  else return {};}
```

Figure 7. Insertion and Find for lazy lists [30]

A List code

Following up on the example from Section 1.1 Figure 7 shows the code for insert and find.

B Proof of Theorem 3.1

Proof. We show that regardless of the number of times T is run, there is a single linearizable instance of each of its instructions that takes effect, and all other instances of each instruction of the thunk have no observable effect.

Let O_k be the kth operation in T's code. We show by induction that all instances of a given operation O_k of T return the same value, across all runs of T in an execution. As the base case, note that all processes that execute T start with the same local variables, and T takes no arguments. Therefore, they begin the execution in the same state. Assume all instances of each operation $O_1 \ldots O_{k-1}$ return the same value, and consider O_k. Since all previous operation instances returned the same value across all runs, O_k is called with the same arguments across all runs. Note furthermore that if O_k executes line 34, the CAS on that line is successful in exactly one instance; all processes executing O_k use position k to access the log, and no process executing a different operation uses position k. Therefore, before the first execution of line 34 for O_k, `log[k] = empty`. Since we assume `empty` is never written in any allocated variable, the new value of the CAS on line 34 will never be `null`. Therefore, the first instance of that CAS will be successful, and all others will fail. Therefore, if O_k is a load or an allocate, since those operations return the value read from `log[k]` after the first CAS on line 34 for O_k, all its instances will return the same value. Note that all other operations do not return a value, so the claim holds.

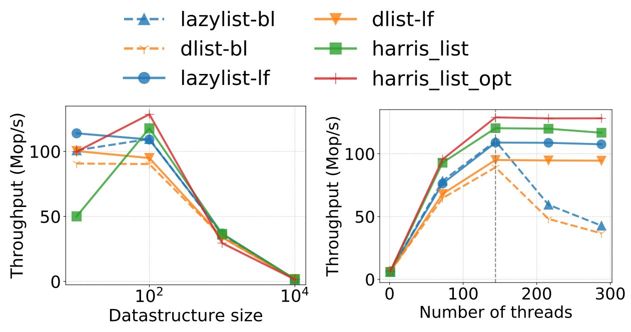
We define *meaningful* instances of each operation O_k: if O_k is a load, the meaningful instance is the first to execute line 34. If O_k is a store or a CAM, there are two meaningful instances: the first that executes the CAS (on line 46 for a store and line 50 for a CAM), and the first to execute line 34.

We can now show that there is exactly one meaningful instance for each operation of T that is linearizable with respect to the rest of the execution, and all others have no observable effect. Note that meaningful loads return the value that they read on line 42. Therefore, they linearize at that line. All other load instances do not modify the memory. Since there is exactly one meaningful instance of each load operation, the claim holds for loads. Since allocate and retire operations are no-ops in the sequential specification, and do not modify memory outside of the system allocate and retire calls, the claim holds for them as well.

We now consider the store operations. There are up to two meaningful stores; either the first that executes line 34, or the first that executes line 46. We name them M_{commit} and M_{CAS} respectively. Note that since all instances of the same operation have the same arguments, and all variables are ABA-free, only the first CAS on line 46 of any operation O_k can be successful. All others will fail. If that CAS is successful, then we linearize M_{CAS} at the CAS. Otherwise, there must have been some other operation, not associated with T, that modified the variable on which O_k operates between when M_{commit} read that variable on line 42, and the time M_{CAS}'s CAS was executed. So, we linearize M_{commit} immediately before the variable it operated on changed. This time is during its interval, since M_{commit} did not yet execute its CAS of line 46 when this occurred. Furthermore, recall that stores may not race with CAM operations. Therefore, the operation that changed this variable must have been a store, which linearized at the point at which it changed the memory (since its CAS must have succeeded), after we linearize O_k. So, O_k can be viewed as having linearized and immediately overwritten. Note that since all other instances of the CAS on line 46 fail by the arguments above, no other instance of this store changes anything but the log.

For CAMs, there are up to two meaningful instances, just like for the store; we similarly name them M_{commit} and M_{CAS}. However, in some cases, the CAS on line 50 is not executed. In that case, there is just one meaningful instance, and we linearize it at line 34. Note a CAM executed on the specified memory location at that point in time would fail, so the effect aligns with its sequential specification. Otherwise, we linearize at line 50 of the M_{CAS}.

Therefore, for any execution E, we can pick out a subexecution E' corresponding to the meaningful instances of each operation of T, with stores and CAMs being picked as described above. E' has exactly one copy of each operation of T in E. Furthermore, all operations of T in E outside of E' have no observable effect, and the effect of E' is the same as if T were executed just once in E. Therefore, T is idempotent. □



(a) 144 th., 5% up., $\alpha = 0.75$ **(b)** 100 keys, 5% up., $\alpha = 0.75$

Figure 8. Throughput of singly and doubly-linked lists.

C Proof of Theorem 4.2

Proof. A *helping chain* is the nested sequence of tryLocks that a process helps. In particular when first helping we add the helped tryLock to the head of the chain, and if while helping the process recursively helps, it is added to the end of the chain. We say a tryLock helps another if either is runs it in line 22 or it succeeds in the test in line 10, but fails in the test on line 16 lock. In the second case the tryLock is said to help the last tryLock on the lock when failing on the test on line 16. Note that this last tryLock will enter after test on line 10 and complete before exiting since it will be unlocked when exiting. Also note that a tryLock will either succeed or add one more to its helping chain.

A helping chain cannot contain two tryLocks with the same owner. This follows from the fact the precedence graph on the locks is lock-cycle free, and due to idempotence the helper and the helpee must follow the same path of helping. Since either the helper or helpee could stall at any point, neither can quit early. Therefore once a tryLock initiated by a particular process p is reached by a process p' while helping, either p completes its attempt and all its helping before p' does, in which case p' will just follow p's path doing the same thing, or p completes after in which case p' will never need to help p again since p is busy. In the first case seeing the same process again would imply a cycle in the precedence graph. This shows p' will only help at most one tryLock owned by each other process. Hence the helping chain has length bounded by p, and its step count is bounded by $O(p\tau)$.

Furthermore note that the last tryLock in a helping chain must complete an embedded (nested) leaf tryLock since otherwise it would fail on a lock and add one more to its helping chain, a contradiction. Therefore each root tryLock is responsible for completing a leaf tryLock, however there is the potential for double counting (e.g. two root tryLocks on the same process could be responsible for completing the same leaf tryLock). We note that when done with a helping chain all tryLocks on it have exited (since we remove the locks when done helping). Therefore the only processes that could follow part of the same chain are ones that have started before the helping chain is done. There can only be $p - 1$ such processes so only that many can be responsible for completing the same leaf tryLock. This shows that every at most p root tryLocks completes a unique leaf tryLock. □

D Linked List Experiments

Experimental results for doubly and singly linked lists can be found in Figure 8. `harris_list` represents Harris's lock-free linked list [28], and `harris_list_opt` represents an optimized version where `find` operations do not perform any helping [15].