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.

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 as fast or faster [1, 17, 32, 54]. 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 [17]. Our experiments verify this (e.g. see the right side of Figure 5d and 5g. 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 [13, 22, 30, 31, 43, 54] and others use fine-grained locks [4, 11, 21, 32, 36, 39, 41, 44]. 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 [52] 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 [23, 28, 31, 53], other multiword operations [13], software transactional memory [25, 46, 48], and specialized data structures [6, 20, 22, 47, 55].

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.

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 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, 12, 18, 19]. 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 them 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 by up to 2.4x.

Our contributions include the following:

1. We present a new practical approach to achieving idempotence in general code, which relies on logging rather than context saving.
2. We present a new approach to lock-free try-locks. They can be nested.
3. We develop a general library-based 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

```
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.

example uses optimistic fine-grained locks [36, 37]. Our C++ code using FLOCK is given in Algorithm 1. Each link holds a key and value, a previous and 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[1]. 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 expression containing the code to be run when the lock is acquired. If the lock is free, try_lock acquires the lock, runs the thunk, releases the lock, and returns the thunk's return value (a boolean). 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[2]. 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, 27, 47, 50]. 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

---

[1]The underscore in mutable_ is used since unforunately mutable is an obscure reserved symbol in C++.

[2]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).

following atomic primitives: *read, write* and *compare-and-swap (CAS)*, defined in the standard way. We also assume a *sysAlloc* which returns an unused block of memory and a *sysRetire* which delays freeing the memory block until it is safe. Each process also has access to private memory.

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 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.

A data structure 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 a data structure specifies code for processes to run for each of its operations. An implementation of a data structure D 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.

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. Some code is naturally idempotent. For example, 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.

In general, however, 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 code idempotent. Several approaches in the literature have shown how to do so [3, 5, 7, 52]. 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 [34]. Note that any procedure with given arguments can be made a thunk by wrapping it in code that reads its arguments from memory.

We follow the definition of idempotence introduced in [5]. A *run* of a thunk T is the sequence of steps on shared data 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. We say a sequence of steps S is *consistent* with a run r of T if, ignoring process ids, S contains the exact same steps as r. We use $E \mid T$ to denote the result of starting from an execution E and removing any step that does not belong to a run of the thunk T.

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 on shared data, there exists a subsequence E' of $E|T$ such that:*

1. *if there is a finished run of T (response on T), then the last step 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 consistent with a single run of T.*

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 step executed for T, the rest are not effectual (i.e. have no impact on the rest of the execution). Furthermore, after one run of T completes, no other runs of T can execute an effectual step.

We assume that a thunk may have *thunk-local* memory which can only be accessed by processes executing the thunk. In our simulation the log is thunk-local. Such memory is not "shared data" as defined in Definition 1.

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 struct, 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 shared<T> indicates a variable of type T that is shared among processes. The log, however, is *thunk-local*; any process executing this thunk uses the same descriptor struct, so the log is shared by all processes that execute this thunk,[3] 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* using the memory primitives *read, write, cas, sysAlloc* and *sysRetire*. 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. 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*. For our purposes a value is *non-mutable* (constant) if it is written once (e.g. on initialization) and only read after it is written.

The idea of the approach is that each process keeps track of its current log (line 5) and how many items it has logged in it 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.

Values are stored in the log using a helper function called *commitValue* (line 31). 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

---

[3]Note that this differs from distributed logs used in, for example, optimistic transactional memory [37], where each process has its own log. It also differs from logs used to commit successful transactions.

```
1   type Log = shared<entry>[logSize];
2   type Thunk = function with no arguments returning bool

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

8   struct descriptor:
9     Log* log;
10    Thunk thunk;
11    mutable<boolean> done;

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

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

21  bool run(descriptor* D):
22    Log* old_log = log; // store existing log and position
23    int old_pos = position
24    log = D->log; // install D's log
25    position = 0;
26    bool returnVal = D->thunk(); // run thunk
27    log = old_log; // reinstall previous log and position
28    position = old_pos
29    return returnVal;

31  <V, bool> commitValue(V val):
32    if (log == null): return <val, true>;
33    bool isFirst = log[position].CAS(empty, val);
34    V returnVal = log[position].read();
35    position++;
36    return <returnVal, isFirst>;

38  struct mutable<V>:
39    shared<V> val;
40    V load():
41      V v = val.read();
42      return commitValue(v).first;
43    void store(V newV):
44      V oldV = load();
45      val.CAS(oldV, newV);
46    void CAM(V oldV, V newV):
47      V check = load();
48      if (check != oldV): return;
49      val.CAS(oldV, newV);

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

57  void retire<V>(V* obj)
58    <_, isFirst> = commitValue(1);
59    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). The log is of fixed sized, but could grow by adding blocks as needed (see Section 6 for details).

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 32). With our locks this 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.

To load a value from a given mutable variable, a process simply does a read from the location, and then tries to commit its value to the log by calling `commitValue`. The return value of the load (line 34) 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 variable, 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 value based on random noise in the processor, this needs to be committed so all instances of the thunk agree on it.

We now show that our idempotence construction is correct; that is, the mutable type implemented in Algorithm 2 is linearizable, and any thunk that wraps all its mutable shared variables with the `mutable` type is idempotent. We begin by outlining a proof of idempotence. For the following theorem, we relax Definition 1 so that retire operations in E' are allowed to appear later than they would have in a single run of T. This has no effect on correctness and at worst it delays the reclamation of memory. Our idempotence construction requires this relaxation because a process can go to sleep before peforming the sysRetire on line 59, and in the meantime, other processes can perform future operations of the thunk, making the retire appear out-of-order.

Theorem 3.1. *Replacing each mutable shared variable accessed by a thunk T with a* `mutable` *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 relaxed definition of idempotence is given in the full version of our paper [8]. 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. □

Idempotent by itself does not guarantee that we are not over-allocating or double freeing. To prevent memory leaks, every block of memory allocated on line 52 that does not get committed to the log is freed on line 54. We also use the shared log to ensure that each object is retired no more than once on line 59.

To complete the correctness proof, we also need to show that `load`, `store`, and `CAM` are linearizable in executions where each instance is run only once. Intuitively, this is

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

5   type Lock = mutable<lockDescr>;

7   bool runAndUnlock(Lock* lock, lockDescr descr):
8     bool result = run(descr.d);
9     descr.d->done.store(true);
10    lock->CAM(descr, lockDescr(descr.d, false));

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

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

Algorithm 3. Idempotent TryLock

because in the absence of repeated runs, the load operation simply reads and returns the variable val, and the store and CAM operations simply read val and try to update it with a CAS. This is a well-known linearizable implementation load, store, and CAM/CAS using just load and CAS, and it is linearizable as long as stores and CAMs do not race.

As mentioned, most previous approaches to idempotence have been based on *context saving* [3, 5, 7, 10, 52]. 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 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 16) and tags it to mark that it is locked (line 17). 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 21) to check if successfully acquired. If acquired or if previously acquired and now done, it runs the code and releases the lock (line 22). If not acquired but currentDescr is locked, then the algorithm helps the descriptor on the lock and unlocks it (line 24). Whether the CAM was successful or not, myDescr needs to be retired (line 25). If on line 15 the lock is already locked, then the algorithm helps the descriptor on the lock and unlocks it (line 26). 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 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 other 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 is run by the runAndUnlock method which can be called on line 22 by the process that installed the descriptor, or on lines 24 or 26 by the helping processes. 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 lock was already taken on line 14 (i.e. the check on line 15 fails) or the attempt to install a descriptor was unsuccessful on line 18 (i.e. the check on line 21 fails), then the tryLock fails and returns false. Otherwise, its descriptor was successfully installed, and it returns the result of running that descriptor on line 22. Note that it is important to check the descriptor's done flag on line 20 because even when the descriptor is successfully installed on line 18, the load on line 19 might not see it because it might have been helped and replaced by another process. Checking the done flag ensures that the tryLock will always return the return value of the descriptor it installed if its CAM on line 18 is successful. □

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. For this purpose we make some assumptions. Firstly, we assume the locks have a partial order $<_p$, and that when nesting locks they are acquired in decreasing order. This is a relatively standard assumption for lock-based algorithms since it prevents lock-cycles and deadlock. In lists and trees, the ordering can be implied by the ordering in the tree. Secondly, we assume that each tryLock includes at most one other tryLock directly inside of it. Note that this still allows arbitrary depth of nesting since the one inside can itself contain another lock inside it.[4] We refer to locks that satisfy these two conditions as *simply nested*. As with some other lock-free mechanisms [25, 48] we also assume the number of locks (or in their case, memory locations) is bounded. We say that a simply nested tryLock *recursively succeeds* if it acquires its lock as well as all locks nested inside of it. Note that for a tryLock a if any one tryLock nested in it recursively succeeds then they all do, including a.

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

---

[4]We expect this requirement is not necessary, but our proof relies on it and it is true for all our tryLock-based data structures.

its critical region. We count all functions in the idempotent interface as unit cost plus the cost of 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.

Theorem 4.2. *Consider an algorithm using simply nested tryLocks for which the maximum step count for any tryLock, not including helping, is bounded. In such an algorithm, a tryLock, including any helping it does, will run in bounded steps, and for every bounded number of tryLock attempts at least one top-level tryLock will recursively succeed.*

Proof. We say a tryLock a with a nested (not necessarily directly nested) tryLock b *helps* another tryLock c if (1) line 24 or line 26 of b runs the thunk installed by c or (2) currentDescr on line 19 of b is unlocked and belongs to c. In the second case, a does not actually help c, but c must have acquired its lock after line 14 of b and released it before line 19 of b so we can give a credit for helping. Importantly, by this definition, if a tryLock performs no helping, then it recursively succeeds. Due to idempotence and simple nesting of locks, every tryLock helps at most one other.

Now note that if a helps b due a conflict on lock l_1 and b helps c due to a conflict on lock l_2 the tryLock that attempts l_2 is nested inside the tryLock that acquired l_1. Since simply nested locks are acquired in the partial order $>_p$, we have that $l_1 >_p l_2$ and more generally that locks decrease along any chain of helping. Assuming a bounded number of locks, the chain will have bounded length and end with a recursively successful tryLock. Hence, running any tryLock takes bounded steps (including helping). Furthermore, since there are a bounded number of locks on the chain, the number of tryLocks responsible for completing the last one is also bounded. Finally we note that although the last one might not be top-level, the fact it recursively succeeds implies the top-level tryLock that contains it recursively succeeds. □

This theorem indicates that simply nested tryLocks are lock-free in that a tryLock must succeed 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 simply nested tryLocks is lock-free. In an algorithm based on optimistic 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, for example, 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 an optimized version of such a strictLock and compare it to the tryLock in Section 8. We note that this implementation of strict locks is not simply nested so is not covered by Theorem 4.1. However, it should be possible to adapt TSP's proof [52] to show that strict locks are lock-free.

5   Related Work

As mentioned, the idea of lock-free locks was introduced by Turek et al [52] and Barnes [3]. The idea of helping dates back earlier, at least to Herlihy's work on wait-free simulations [33]. 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. [15].

The idea of *idempotence* has been used in the literature a variety of contexts [7, 10, 10, 12, 18, 19, 35, 40]. Kruijf, Sankaralingam and Jha [19] give a nice overview although only up to 2012. More recent work has focused on using idempotence for fault tolerance (e.g., [7, 10, 40]). 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 [51] 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 [42] 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 [30] and Natarajan and Mital's BST [43], 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 [13] 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 [23, 28, 31].

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 [45]. We assume a process can fail or stall at any instruction. Gidenstam and Papatriantafilou [26] 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 [24, 29]. 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 [32] (lazylist), an adaptive radix tree [38, 39] (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), 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, and an (a,b)-tree (abtree). To support concurrent accesses, the data structures use fine-grained, optimistic locking, as in [11, 32, 36, 39]. 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 implement a tryLock and a strictLock version of each data structure. Both tryLock and strictLock can either be lock-free (with helping and logging) or blocking (using test-and-test-and-set locks), and with our library, this choice can be made by changing a flag 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. Our implementations of these optimistic, fine-grained locking data structures are available at https://github.com/cmuparl ay/flock.

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.

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 hyper-threaded 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. We used ParlayLib [9] for scalable memory allocation.

Workloads. We experiment with set data structures supporting `insert`, `delete` and `lookup` with 8-byte keys and 8-byte values. Our experiments follow a similar methodology to previous papers [1, 17]. 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 `inserts` and `deletes`, keeping the data structure size stable throughout the run. Each experiment is run for 3 seconds (sufficient for reaching a stable state) and repeated 4 times. The first run is a warmup run and an average of the last 3 runs is reported. Before the warmup run, we shuffle the ParlayLib memory allocator by allocating a large number of nodes and freeing them in a random order to increase consistency across runs. Standard deviation between runs is small enough that the error bars in our graphs are only visible for a small number of data points.

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 [16]. We mostly run with 5% and 50% updates, following YCSB Workloads B and A, respectively.
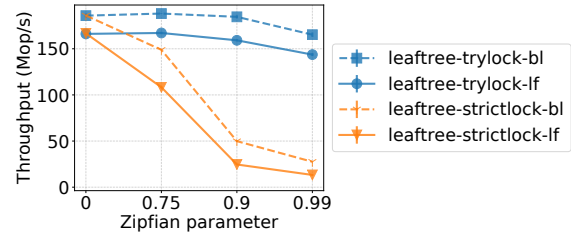
Figure 4. Comparing try lock with strict lock on workload with 100K keys, 144 threads, and 50% updates. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.
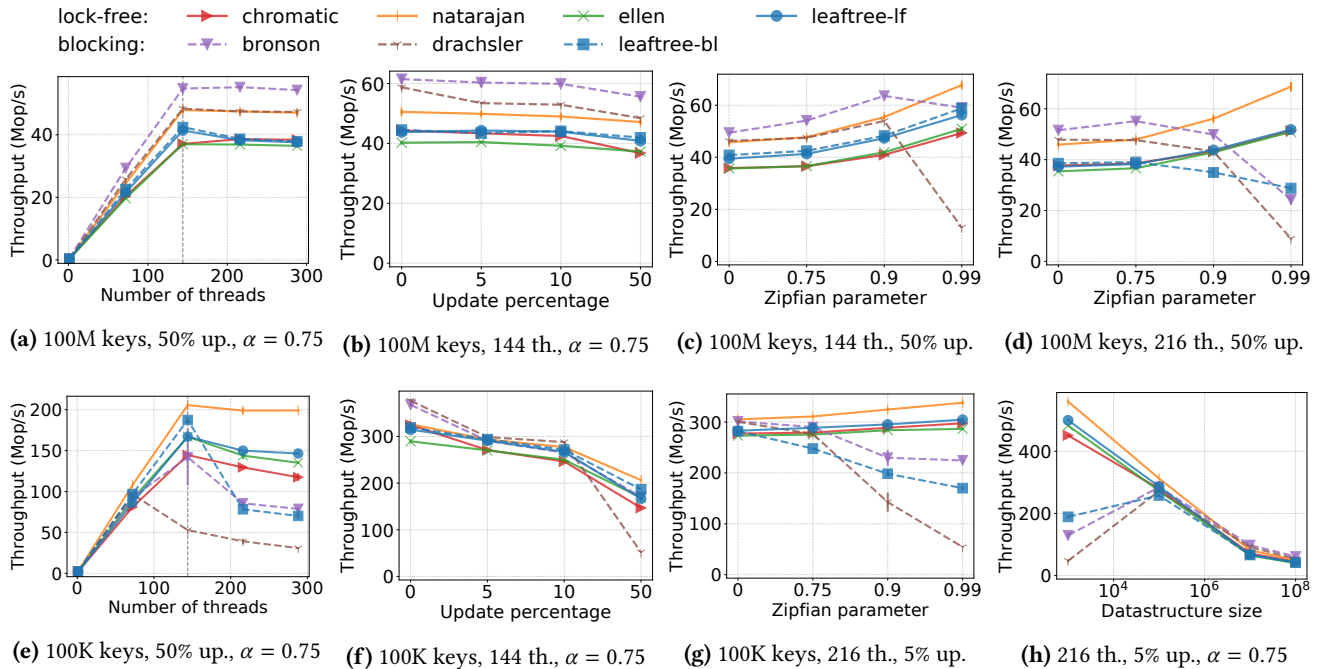
Our experiments vary four parameters: data structure size, update rate, α, and number of threads. We show graphs along each of the dimensions, fixing the other three. Since `arttree` is a trie data structure, it benefits heavily from densely packed keys, so we sparsify the key range by hashing each key from $[1, r]$ to a 64-bit integer. This does not affect the other data structures since they either are purely comparison based or hash the keys themselves.

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 report on tryLocks.

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 [11], Drachsler [21]) and lock-free (Ellen [22], Chromatic [14] and Natarajan [43]) binary search trees. These implementations were obtained from the SetBench benchmarking suite [1]. Bronson and Chromatic are the only balanced tree among these implementations. Regarding the lock-free trees, Ellen and Natarajan are implemented directly from CAS whereas Chromatic is implemented using the higher-level LLX/SCX primitives [13]. Note that in all the graphs, lock-based algorithms are denoted by dotted lines and lock-free algorithms appear as solid lines.

Figures 5a- 5d consider the case where the tree does not fit in cache and Figures 5e- 5g consider the case where they do. In out-of-cache workloads, performance is dominated by cache misses incurred during the traversal phase. Figure 5b shows that the cost of updating the tree is small compared to these cache misses, whereas in Figure 5f, increasing the

(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'. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.

(a) 100M keys, 50% up., $\alpha = 0.75$ **(b)** 100M keys, 216 th., 50% up.
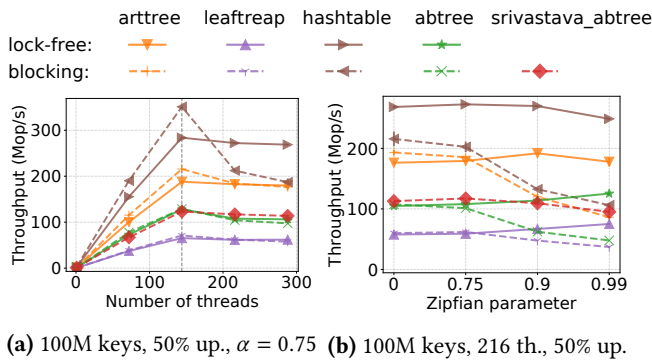
Figure 6. Throughput of concurrent set data structures.

percentage of updates significantly reduced throughput. All trees scale well, up until oversubscription (Figures 5a and 5e), with the exception of Drachsler in Figure 5e. Bronson is generally the fastest when tree size is large because it is better balanced compared to the other trees (many of which are only balanced in expectation due to random inserts), resulting in shorter traversals and less cache misses. As the zipfian parameter α increases, all trees except Bronson and Drachsler 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 blocking strict-locks, this extra contention

out-weighs the benefits of locality. This effect is even more severe for small trees (Figure 5g).

Lock-free vs blocking. Next, we compare the throughput of lock-free data structures with blocking ones, with particular emphasis on `leaftree-lf` and `leaftree-bl`, the lock-free and blocking variants, respectively, of our `leaftree`. 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. This overhead is only visible in small trees with high update rates (Figures 4 and 5e). Across all the graphs in Figure 5, the overhead of using lock-free locks rather than traditional blocking locks is no more than 11%. Furthermore, most graphs do not show any visible overhead.

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 the right side of Figures 5d and 5g and the left side of Figure 5h where the four lock-free trees outperform the three blocking trees.

(a) 144 th., 5% up., $\alpha = 0.75$ **(b)** 100 keys, 5% up., $\alpha = 0.75$
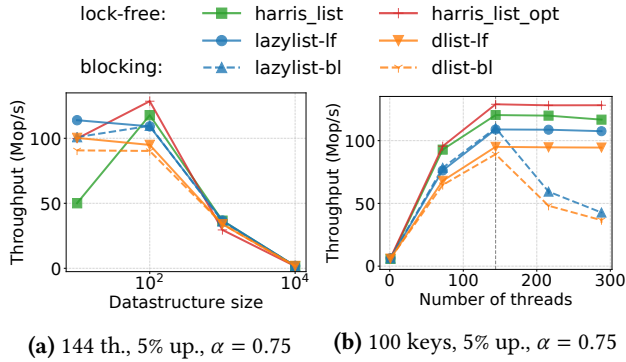
Figure 7. Throughput of singly and doubly linked lists. The 'bl' and 'lf' suffixes represent the blocking and lock-free version of our locks, respectively.

In particular, `leaftree-lf` outperforms `leaftreap-bl` by up to 2.4x in Figure 5h.

Other set datatypes. In Figure 6, `arttree`, `leaftreap`, `abtree` and `hashtable`, generally follow the same pattern as `leaftree`. That is, lock-free versions outperform their blocking counterparts in oversubscribed, high contention scenarios (right side of Figure 6b), by up to 2.5x in the case of the `hashtable` and 2x for the `arttree`. In non-oversubscribed scenarios (left size of Figure 6a), the overhead of using lock-free locks is small, especially for `abtree` and `leaftreap`. The overhead of lock-free locks is highest in the `hashtable` because its search time (i.e. fraction of time spent outside of the critical section) is small and hence the overhead for the locked part plays a larger role. Figure 6 also compares our data structures with Srivastava's CoPub-ABtree [49], a state-of-the-art blocking (a,b)-tree. Our lock-free `abtree` performs similarly to `srivastava_abtree` in most cases but is up to 32% faster at the right of Figure 6b.

Linked List Experiments. Figure 7 compares doubly and singly linked lists written using our lock-free locks (`dlist` and `lazylist`, respectively) with Harris's lock-free singly linked list [30] (`harris_list`), and an optimized version of Harris's list where `find` operations do not perform any helping [17] (`harris_list_opt`). In most cases, our lock-free `lazylist` is slower than `harris_list_opt` by about 16% because the descriptors in `harris_list_opt` are optimized to simply be flags. Interestingly, the lock-free versions of `dlist` and `lazylist` outperform their corresponding blocking versions even without oversubscription on small lists (left of Figure 7a).

The pseudo-code for `dlist` was presented earlier in Algorithm 1 and Figure 7 show that this simple algorithm performs well. The overhead of maintaining back pointers is only about 13% (comparing `dlist` with `lazylist`).

9   Conclusion

We presented a mechanism for implementing lock-free locks, and a library-based implementation. It is the first such library implementation of lock-free locks we know of. The approach is practical in two senses. Firstly, in terms of performance it is competitive with state-of-the-art lock-free and lock-based data structures. Secondly, using the library requires very few changes to existing lock-based implementations—basically wrapping shared values in a mutable, and using the Flock lock structure and memory management. In terms of functionality it significantly extends previous suggested approaches to lock-free locks, supporting memory management, races, and tryLocks.

We separate out the idea of idempotent blocks of code (thunks) and present a general and efficient approach along with a C++-based library to support them. The approach supports arbitrary code with load, stores and CAMs on shared locations, as well as memory allocation and retirement from a shared pool. A thunk using the approach can be run any number of times with instructions interleaved in any way while behaving like it ran once. The approach uses a shared log for each thunk so that separate runs of the thunk see the same result. The idempotent construction could be of independent interest.

We implemented several data structures using the approach. With regards to the opening question of whether to be lock-free or not to be, the experiments clearly indicate the advantage of lock-freedom when processors are oversubscribed. Our experiments are some of the first on concurrent data structures that study this effect. The experiments also show that the overhead of being lock-free for our structures is relatively small (rarely more than 10%) and often hardly noticeable.

Acknowledgments

References

[1] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In *USENIX Annual Technical Conference*. 295–306.

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

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

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

[5] Naama Ben-David and Guy E. Blelloch. 2021. Fast and Fair Lock-Free Locks. arXiv:2108.04520 [cs.DC]

[6] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *International Symposium on Distributed*

Computing (DISC). 12:1–12:20.

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

[8] Naama Ben-David, Guy E. Blelloch, and Yuanhao Wei. 2022. Lock-Free Locks Revisited. arXiv:2201.00813 [cs.DC]

[9] Guy E Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib-A toolkit for parallel algorithms on shared-memory multi-core machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 507–509.

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

[11] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP).* 257–268.

[12] Jeremy Brown, J. P. Grossman, and Tom Knight. 2002. A Lightweight Idempotent Messaging Protocol for Faulty Networks. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* 248–257.

[13] Trevor Brown, Faith Ellen, and Eric Ruppert. 2013. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing (PODC).* 13–22.

[14] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP).* 329–342.

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

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

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[18] Marc de Kruijf and Karthikeyan Sankaralingam. 2013. Idempotent Code Generation: Implementation, Analysis, and Evaluation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO).*

[19] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In *ACM Conference on Programming Language Design and Implementation (PLDI).*

[20] Dave Dice and Alex Garthwaite. 2002. Mostly lock-free malloc. In *International Symposium on Memory Management (ISMM).* 163–174.

[21] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP).*

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

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

[24] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

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

[26] Anders Gidenstam and Marina Papatriantafilou. 2007. LFthreads: A Lock-Free Thread Library. In *Conf. on Principles of Distributed Systems*

(OPODIS).

[27] Michael Greenwald. 2002. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *ACM Symposium on Principles of Distributed Computing (PODC).* 260–269.

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

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

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

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

[32] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-Based Set Algorithm. In *Conf. on Principles of Distributed Systems (OPODIS).*

[33] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (jan 1991), 124–149.

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

[35] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. 2006. Exploiting Reference Idempotency to Reduce Speculative Storage Overflow. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 5 (sep 2006), 942–965.

[36] H. T. Kung and Philip L. Lehman. 1980. Concurrent Manipulation of Binary Search Trees. *ACM Trans. Database Syst.* 5, 3 (1980).

[37] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226.

[38] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *IEEE International Conference on Data Engineering (ICDE).*

[39] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proc. International Workshop on Data Management on New Hardware.*

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

[41] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *ACM European Conference on Computer Systems (EuroSys).*

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

[43] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP).*

[44] William Pugh. 1989. *Concurrent Maintenance of Skip Lists.* Technical Report TR-CS-2222. Dept. of Computer Science, University of Maryland, College Park.

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

[46] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A wait-free persistent transactional memory. In *IEEE/IFIP International Conference on Dependable Systems and Networks*

(DSN). 151–163.

[47] Niloufar Shafiei. 2014. Non-Blocking Doubly-Linked Lists with Good Amortized Complexity. arXiv:1408.1935 [cs.DC]

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

[49] Anubhav Srivastava. 2021. *Extremely fast (a, b)-trees at all contention levels.* Master's thesis. University of Waterloo.

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

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

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

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

[54] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. *ACM SIGMOD International Conference on Management of Data (SIGMOD),* 473–488.

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

A Artifact Evaluation Appendix

A.1 Abstract

This artifact contains the source code and scripts to reproduce all the graphs in Section 8. It includes an implementation of the FLOCK library described in Section 6 as well as all the data structures described in Section 7.

A.2 Artifact check-list (meta-information)

- **Algorithm:** FLOCK library plus the optimistic fine-grained locking data structures described in Section 7.
- **Program:** microbenchmarks
- **Compilation:** g++ 9.3.0
- **Binary:** binary not included
- **Run-time environment:** Ubuntu 16.04.6 LTS
- **Hardware:** Multi-core machine, preferably with at least 64 logical cores
- **Output:** graphs from Section 8 as PNG files.
- **Experiments workflow:** one script for compiling the experiments and one script for generating all the graphs.
- **Disk space required (approximately):** 60 MB
- **Time needed to prepare workflow:** approximately 5 minutes
- **Time needed to complete experiments:** approximately 6 hours
- **Publicly available:** yes
- **Code licenses:** MIT License

A.3 Description

A.3.1 How delivered. Available as open source in the ae branch of the following GitHub repository: https://github.com/cmuparlay/flock.

A.3.2 Hardware dependencies. To accurately reproduce our experimental results, a multi-core machine with at least 64 logical cores is recommended.

A.3.3 Software dependencies. Our artifact is expected to run correctly under a variety of Linux x86_64 distributions. Our experiments were compiled using g++ 9. For scalable memory allocation in C++, we used jemalloc 5.2.1 (https://github.com/jemalloc/jemalloc/releases/download/5.2.1/jemalloc-5.2.1.tar.bz2) as well as the ParlayLib memory allocator, which is included with the artifact. Our scripts for running experiments and drawing graphs require a Python 3 installation with mathplotlib. We used the numactl command to evenly interleave memory across the NUMA nodes.

A.3.4 Data sets. None.

A.4 Installation

Source code can be complied by running bash compile_all.sh.

A.5 Experiment workflow

After compiling, run `./generate_graphs_from_paper.sh` to generate all the graphs and store them in the `graphs/` directory.

A.6 Evaluation and expected results

On a machine with 140 or more logical cores, the results should be very similar to those reported in this paper.

A.7 Experiment customization

For instructions on how to customize the number of threads, workload, and data structure size in each experiment, please see the README file in the ae branch of the GitHub repository (https://github.com/cmuparlay/flock).

A.8 Notes

None.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://ctuning.org/ae/submission-20190109.html
- https://ctuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging