Abstract

Data-structures can benefit from dynamic data layout modifications when the size or the shape of the data structure changes during the execution, or when different phases in the program execute different workloads. However, in a modern multi-core environment, layout modifications involve costly synchronization overhead. In this paper we propose a novel layout lock that incurs a negligible overhead for reads and a small overhead for updates of the data structure. We then demonstrate the benefits of layout changes and also the advantages of the layout lock as its supporting synchronization mechanism for two data structures. In particular, we propose a concurrent binary search tree, and a concurrent array set, that benefit from concurrent layout modifications using the proposed layout lock. Experience demonstrates performance advantages and integration simplicity.

Keywords Synchronization, Concurrent Data Structures, Data Layout

1. Introduction and Motivation

With the proliferation of multi-core machines, the need for high performance concurrent data structures is becoming acute. Much work has studied avenues for making concurrent data structures more performant and responsive. But some techniques for improving data structure performance are still limited to sequential data structures as their complexity in a concurrent setting seems too high. One such technique is layout modification (Wael et al. 2015; Wael 2015; Xu 2013; Sung et al. 2010; Li et al. 2013), which allows modifying the underlying structure of the data to achieve better performance when program behaviors evolve during the execution. Such layout changes are used in various scenarios, e.g., by compilers when parts of the program make limited use of data (Kennedy and Ziarek 2015; Kandemir et al. 1999a; Mannarswamy 2009; Ding and Kennedy 1999), improving data locality and reducing false sharing through data type layout modification (Chakrabarti and Chow 2008; Raman et al. 2007; Eizenberg et al. 2016) or in order to improve matrix multiplication performance (Kandemir et al. 1999b; Lu et al. 2004), etc. But all these uses are limited in existing concurrent data structure designs due to the seemingly high complexity of designing a concurrent layout modification and the cost of synchronization.

In this paper we propose a paradigm and a synchronization mechanism for layout modification in a concurrent setting. The new synchronization paradigm (and its supporting mechanism) is called layout lock and it enables layout changes during concurrent execution. The paradigm involves three types of operations executed on the data structures: read, write, and layout change. Readers perform read-only accesses; writers may read and update data; and layout changers can modify the internal representation of the data.

Adding a layout lock to an existing data structure implementation requires some care. First, the read operation is executed optimistically, and so a mechanism for handling failed reads must be devised. Second, care must be taken by the layout changer to ensure that (failing) optimistic reads do not reach a bad execution state (e.g., segmentation fault). Finally, in order to obtain performance advantages from layout modifications, one must design data structures for which layout modifications indeed improve performance. In essence, one needs to implement “Just-in-Time” data structures, that can dynamically adapt to usage patterns (Wael 2015; Xu 2013). Using the proposed layout lock typically requires the addition of the appropriate lock acquisitions and releases before and after each of the operations (i.e., read, write, and layout change), and the handling of failed optimistic reads. We discovered that, for data structures that can deal with optimistic reads, adding the layout lock was simple and fast.

Based on the layout lock paradigm, we developed an efficient concurrent binary tree that outperforms known concur-
rent tree implementations on large trees. Our key observation is that nodes in the high levels of a tree (closer to the root) change infrequently compared to nodes closer to the leaves. Based on this observation, the proposed tree packs sets of neighboring nodes in high levels of the tree into small arrays. The packing improves locality and search speed. Interestingly, the design process was surprisingly simple. We first designed and built this efficient tree in the (simpler) sequential setting, and then we added the layout lock mechanism in almost no time to allow concurrent layout change. This yielded a highly scalable and efficient concurrent tree.

In addition, we also study an array-based implementation of a concurrent set, where only part of the set is sorted. Concurrent layout changes allow occasionally merging the unsorted part into the sorted part, thus speeding up searches. The resulting set is extremely simple to design (using the layout lock) yet provides excellent scalability and performance for small to medium sized sets.

The layout lock paradigm focuses on the problem of synchronizing layout changes with other accesses (i.e., with reads and writes). Synchronization between readers and writers is not the main focus of this paper. We assume that readers and writers are synchronized by some state-of-the-art technique and that readers are extremely fast, so that the overhead of additional synchronization makes a difference.

The layout lock implementation that we propose incurs a negligible overhead on reads of the data structure and a small overhead on updates of the data structure. It assumes that typical uses of the layout lock execute layout modifications infrequently (or even rarely); they execute writes more frequently; and they execute reads most frequently. We believe this behavior is typical in practice, and it holds in the examples presented in this paper.

A possible naïve implementation for a layout lock is to use a reader-writer lock (Mellor-Crummey and Scott 1991; Hsieh and Weihl 1992; Lev et al. 2009; Calciu et al. 2013) to coordinate layout changes. In this setting, a layout changer plays the writer role and all other accesses to the data structures (either reads or writes) play the role of the reader in the reader-writer lock. The drawback of this solution is the cost of reading the data structure. Any thread that reads the data structure needs to announce that it is doing so, and typically also execute a memory fence to make sure that other threads are aware of its announcement. Note that this overhead is imposed even when no layout change actually occurs.

Recently, the idea of StampedLock (Lea and JSR-166) was proposed and incorporated into Java 8. The StampedLock is conceptually based on the SeqLock (Lameter 2005; Boehm 2012) used in the Linux kernel. It allows readers to proceed optimistically by letting them record a stamp (sequence number) at the beginning of the operation and compare the recorded stamp to the current stamp at the end. This lets readers run without any (hardware) memory fence.

Our proposed layout lock implementation also offers optimistic reads, but further optimizes their performance for the case of rare layout changes and frequent reads by multiple reader threads. Reads of the data structure, which are typically most frequent, are executed with no interference, except for checking a local flag (after the read) to verify that the read value is correct. This reduces the overhead of synchronization to a negligible cost.

For writers we allow a slightly higher overhead, to make sure they do not interfere with a layout change. Yet the overhead is still small and the write operations are scalable.

Organization: The rest of the paper is organized as follows. We present the layout lock, its interface, and implementation in Section 2. In Section 3 we present two data structures that can benefit from layout modifications. We present performance evaluations of these data structures with the layout lock in Section 4. Related work is discussed in Section 5, and we conclude in Section 6.

2. Layout Lock
The layout lock paradigm and mechanism is a synchronization methodology for allowing layout modifications concurrently with reads and writes of a data structure. A layout change is a significant modification to the data representation, such as moving the data (content) to a different location in memory, or changing the way the data is organized.

2.1 Paradigm and Interface
The layout lock interface is similar to Java’s StampedLock (Lea and JSR-166) interface and bears a resemblance to the well-known reader-writer lock (Mellor-Crummey and Scott 1991; Hsieh and Weihl 1992; Lev et al. 2009; Calciu et al. 2013). That is, the lock distinguishes between the different types of participants by using separate access methods for each type of participant. The standard reader-writer lock recognizes two types of participants: a reader and a writer. The layout lock interface acknowledges three types of participants: a reader, a writer and a layout changer, and provides a different set of access methods for each type accordingly. Each of the participants may acquire or release the lock appropriately, yet, the layout lock interface diverges from the standard locking interface for the readers. In particular, releasing a read lock returns a boolean value that indicates the validity of the data that was read. If the data is invalid, the read operation may be retried.

A writer operation may contain several accesses to the data structure, some of them can be reads and other can be updates of the data structure. A reader operation may execute several actual read accesses to the data structure, but it does not update the data structure. A reader operation should have no effect visible by other threads.

Semantically, the layout lock provides guarantees that are similar to a read-write lock, where the layout changer plays the role of the writer (of the read-write lock) and the
readers and writers play the role of the reader (of the read-write lock). Specifically, it is guaranteed that read and write operations do not occur concurrently with a layout change and that two layout change operations do not overlap. The writer operation appears to happen either before or after the layout change takes place. For readers, it is guaranteed that a reader operation succeeds (i.e., returns TRUE) only if the read operation can be viewed as happening before or after a layout change, but not concurrently.

Combining the layout lock with other synchronization methods (e.g., between readers and writers) while avoiding deadlocks, requires keeping a strict order between the synchronization mechanisms. For example, always start by acquiring the layout lock and then acquiring the other lock required for synchronization. In the rest of this paper we ignore any additional synchronization and focus only on the synchronization mechanism for enabling a layout change.

The standard way for readers to use the layout lock appears in Listing 1. The term lock in the code stands for an instance of a layout lock used to protect the data structure.

```java
Listing 1. Reader
1  start:
2   lock.startRead();
3   //execute reader code
4   if(!lock.endRead()) {
5     goto start;
6   }
```

The interface for writers and layout changers is similar to a standard lock. Acquiring or releasing the layout lock in write mode or layout change mode cannot fail. The code is presented in Listing 2 and Listing 3.

```java
Listing 2. Writer
1  lock.startWrite();
2   //execute writer code
3  lock.endWrite();
```

```java
Listing 3. Layout changer
1  lock.startLayoutChange();
2   //execute layout changer code
3  lock.endLayoutChange();
```

### 2.2 Implementation

The proposed implementation starts from any standard (efficient) reader-writer lock, which coordinates the accesses of writers and layout changers. Denote this lock by RW-LOCK. We then add a specialized mechanism to synchronize the readers with the layout changers and facilitate a fast path for them. More specifically, a layout-changer acquires the writer lock of the RW-LOCK, and releases it upon completing the layout change. A writer acquires the reader lock of the RW-LOCK and it releases it after completing the operation. This ensures that writers do not conflict with layout changers and that two different layout changers do not execute simultaneously. It remains to describe the synchronization of readers with the layout change operation. To this end, we assume that it is possible to (quickly) check whether the RW-LOCK is being held in the write mode, which in our case means an active layout change. Such a check can be easily added to reader-writer locks that are available in the literature.

The main idea for obtaining fast readers execution is the use of optimistic execution. A reader executes its reads regardless of whether a layout lock is being held or not. In other words, the startRead implementation is empty. When the lock is released, the reader checks whether a layout change operation may have overlapped the reader execution. If it did, the endRead method returns FALSE signaling that the read value may be stale. However, if it can be determined that no part of the reader operation overlapped with a layout change, then the endRead returns TRUE.

To check whether a layout change might have overlapped with a read operation, a per-thread flag, denoted the dirty flag, is maintained. The dirty flag signifies a possible concurrent layout change execution. A reader on the fast path execution simply reads the dirty flag during endRead and if the dirty flag is off, the reader completes its execution successfully. On the other hand, if the dirty flag is set, then the reader clears the dirty flag. It then checks whether a layout change is still active and if yes it sets again its dirty flag to make sure that it does not return TRUE until the layout change completes. Finally, the reader returns FALSE to signify that the data that was read may be invalid and indicate that the read operation should be retried. The reader can tell that a layout change has completed by checking that the reader-writer lock is no longer being held in the writer mode.

This implementation relies on the reader executing a memory fence when clearing its dirty flag, before checking whether a layout change is currently active. The strict order is required to ensure that the dirty flag is kept TRUE during a layout change. Therefore, if a reader resets its dirty flag and observes an active layout change, it sets the dirty flag back to TRUE. Without the memory fence, the following race is possible: a reader finds that no layout change is active, then a layout change sets the reader’s dirty flag, and finally, the reader clears its dirty flag and continue its execution.

A second important ordering that this implementation relies on, is that reading the dirty flag happens after all the reads in the reader code completed. If the underlying machine has a (very) relaxed memory model, then a read memory fence is required. On standard memory models, such as Total Store Order (TSO) or Partial Store Order (PSO) used with x86 or SPARC, a hardware memory fence is not required. However, a compiler fence is always required to restrict the compiler from changing the order of these reads.

The layout changer first acquires the underlying reader-writer lock in write mode. This prevents writers, who ac-
quired the RW-LOCK in the read mode, and other layout changers from accessing the data structure. Then, before executing the layout change, it sets the dirty flags of all threads as TRUE. From that point and on, readers that have not completed a read operation, must return FALSE. This holds until the layout changer completes and releases the reader-writer lock. The code is presented in Listing 4.

```cpp
Listing 4. layout lock Implementation

class layoutLock{
 ReadWriteLock rwlock;
  bool dirty[MAX_THREADS];

  void startRead(){
  }
  bool endRead(){
    //read memory fence if needed
    if (dirty[tid]) {
      dirty[tid] = false;
      _memory_fence();
      if (rwlock.isWriterActive())
        dirty[tid] = true;
      return false;
    }
    return true;
  }
  void startWrite(){
    rwlock.startRead();
  }
  void endWrite(){
    rwlock.endRead();
  }
  void startLayoutChange(){
    rwlock.startWrite();
    for thread in THREADS {
      dirty[thread] = true;
    }
    _memory_fence();
  }
  void endLayoutChange(){
    rwlock.endWrite();
  }
}
```

2.3 Malformed Pointers and Memory Management

In the layout lock mechanism, in order to achieve better performance, readers are not required to announce their operation. This implies that sometimes readers execute while a layout changer is active, until they find out at the end of the read that the read operation must repeat. However, a read operation may consist of several accesses to the data structure, including reading and dereferencing pointers. This places a restriction that the layout changer must modify the data structure in a way that would allow readers to reach the endRead operation without crashing. For example, the layout changer should install only valid pointers in the data structure. Otherwise, a reader may read and dereference a malformed pointer, causing a crash before reaching the end of the read operation. The layout changer may insert NULL pointers if readers can handle them properly.

**Memory management.** Freeing memory space that the layout changer vacates requires some care. Since readers may still read from the old version of the object, a layout changer cannot immediately free unused memory after the layout change.

The freeing of unused memory can be handled safely in various ways. The easiest is to use automatic memory reclamation (i.e. garbage collection). If garbage collection is not available on the platform, then memory reclamation can be executed using mechanisms designed for lock-free data structures which deal with similar concerns, such as (Cohen and Petrank 2015b,a; Alistarh et al. 2015; Brown 2015). Alternatively, the following adaptation of the epoch-based memory management can be used. Following a layout change, when all dirty flags have been reset, it is possible to recycle memory freed by the layout change as it is guaranteed that none of the threads are accessing stale memory.

3. Layout Change Usage Examples

In this section we first present a novel concurrent tree design denoted DLTree, which uses the layout lock paradigm to build a highly concurrent binary search tree. The tree changes its layout to allow faster access. Next, we present an array-based implementation of a concurrent set, where layout changes allow most contain operations to finish as a simple binary search.

3.1 DLTree: A Fast Concurrent Binary Tree

Concurrent binary trees were extensively studied in the literature (Bronson et al. 2010; Morrison and Arbel 2015; Natarajan and Mittal 2014; Braginsky and Petrank 2012; Drachsler et al. 2014; Ellen et al. 2010). In this subsection we propose a new concurrent binary tree design, called a dynamic-layout tree (DLTree), which employs a layout change and the layout lock paradigm and implementation to obtain high performance. A core observation used in this design is that nodes of low-depth (which are closer to the root) change infrequently, whereas the set of nodes of higher depth (which are closer to leaves) change frequently. We use this observation is by noting that the almost-immutable high levels of the tree can be compacted to provide better cache utilization and faster traversals. The details of this compaction are described in Subsection 3.1.2 below. In what follows, we explain how this all leads to a faster concurrent tree implementation.

The DLTree maintains lower-depth nodes in a compacted form. The compacted form can be traversed quickly but it is hard to modify. Since the tree size changes during the execution, we should be able to change the number of levels we consider “low depth” and so it should be possible to move parts of the tree from non--compacted form to compacted form and vice versa. The DLTree algorithm deals with mod-
3.1.1 Synchronizing Tree Accesses

The two parts of the tree are used to optimize the synchronization on tree access. The backbone is designed to support very frequent reads, and rare layout changes. Thus, it is protected using the layout lock. The treelets receive frequent modifications, but these modifications are typically distributed among the many treelets, thus the treelets rarely need to handle contention. In our implementation we attempt to keep each treelet small and we synchronize each treelet using a simple coarse-grained lock (i.e., one lock per treelet).

While the proposed design is extremely simple, care is needed at the point in which a search moves from the synchronization mechanism of the backbone to the lock of the treelet. To this end, we acquire the lock of the treelet before releasing the read lock of the layout lock. This follows the spirit of the well-known hand-over-hand algorithms. However, doing so breaks the invariant that a read operation of the layout lock does not write to a global state. The read operation writes to the treelet lock while acquiring it. If the release of the read lock (of the layout lock) returns FALSE, then this means that a layout change has occurred and the read values cannot be trusted. Luckily, the treelet lock is not part of the backbone and is not protected by the layout lock. Therefore, we can simply continue by releasing the treelet lock and restarting the whole operation.

While composing the two locks is non-trivial, it is still significantly simpler than using fine-grained synchronization. Interestingly, it took less than half an hour to make the basic sequential tree concurrent, by adding the layout lock and the lock-per-treelet synchronization. This stands in contrast to building any of the known concurrent trees. They all employ sophisticated complex synchronization mechanisms.

3.1.2 Backbone Implementation

The treelets are implemented in a very straightforward manner. Their nodes contain a key, a data pointer, two next (left and right) child pointers, and a type field explained below. The backbone tree structure is more complex and in this subsection we present its implementation. We start with the underlying data structure. The backbone tree is implemented as a complete (perfect) external binary tree, whose depth is divisible by 4. Each set of 15 nodes, starting at a node at level $4i$ and including all its descendant of depth 3, i.e., up to its 8 descendants at level $4i+3$, are compacted into a single object called BigNode. A complete binary tree of depth $4d$ yields a complete tree of BigNodes of depth $d$, where each BigNode has 16 BigNode descendants. The leaves of the backbone tree point to treelets. We only implement a search through the backbone. There are no insert or delete operations to specific nodes, as this structure only serves as a backbone directory into the treelets where the actual insert and remove operations take effect. A modification of the backbone occurs when it needs to be enlarged or shrunk. This modification is executed as a layout change and will be discussed below.

BigNodes are organized in memory with care for locality. A BigNode consists of 15 keys, 16 next pointers, and a type field to distinguish a BigNode descendant from a treelet descendant. The type and the 15 keys reside on one cache line (assuming a 4-byte key and a 64-byte cache line), and the next pointers reside on two additional cache lines (assum-
When enlarging (or shrinking) the tree, we take the opportunity to rebalance the tree. We start by acquiring the layout lock in the layout change mode. Then, we gather all data from all treelets in a sorted vector. This is done by executing the following for each treelet: acquire its lock and move its data to the vector. After gathering all data in a sorted vector, a balanced tree (which is not necessarily full) is built with the existing data. The levels that are closer to the root are built as the backbone, and the rest of the levels create a set of treelets. Finally, the layout lock and the locks of all treelets are released\(^1\). Since the layout change operation occurs rarely, we made no attempt to optimize this procedure. Pseudo-code illustrating an enlargement of the tree appears in the method `enlargeBackbone` in Listing 6 below. The shrinking operation is similar.

### Avoiding Malformed Pointers

Read operations are executed concurrently while layout changes take place. Since a read operation may contain several actual read accesses to the global data, it is possible that some of these accesses will happen after a layout change took place. The data structure algorithm must be aware of this problem and make sure this does not cause a crash. For the tree, a problem may arise if a stale pointer is read and then dereferenced. We resolve this problem by maintaining an invariant that a next pointer of a BigNode points either to a valid treelet or to another BigNode. Thus, when the backbone is enlarged, new BigNodes are added only after their next pointers are populated.

#### 3.1.3 Pseudo-Code for the Tree Operations

The DLTree structure is provided in pseudo-code in Listing 5. The tree consists of two types of objects: treelets and BigNodes. It is possible to distinguish between the two using the type field of the node. A treelet contains a lock and a tree. The BigNode underlying structure is as discussed in Section 3.1.2. Traversing the four layers of a BigNode and returning the next pointer is implemented as the `get` method of the BigNode class. Finding the appropriate pointer to descend on each level of the BigNode involves multiplying the index by two, and then adding 1 if the input key is larger than the node key.

```
1 class base {
2   enum { BIG_NODE, TREELET } type;
3 }
4 class BigNode : public base {
5   keys[15];
6   base *nexts[16];
7   public:
8   base + get(key) {
9     keys = &this->keys[-1];
10    idx=1;
11   idx=2*idx + (key>keys[idx]);
```

\(^1\) Actually, it is possible to simplify this process by releasing the lock of each treelet after copying its values. Arguing that this provides a coherent snapshot of the tree is a bit more involved and left out of this short version of the paper.
The DLTree operations (search, insert, and remove) are presented in Listing 6. Each of the operations start by finding the appropriate treelet to execute the operation, and acquiring the lock for this treelet. Finally, the treelet lock is released. The operation that returns the required treelet after acquiring its lock is called getLockedTreelet.

The getLockedTreelet method searches the backbone to find the relevant treelet. The traversal is executed using the BigNode get method. The entire search is enclosed with the layout lock held in the read mode. The backbone traversal is completed when a non-BigNode node is discovered (thus, a Treelet). The treelet lock is then acquired. Finally, if releasing the layout lock returns FALSE, the treelet lock is released and the operation is restarted. If releasing the layout lock returns TRUE, the locked treelet is returned.

After executing an insert operation (or a remove operation) a heuristic is used to determine if it is beneficial to enlarge (or shrink, resp.) the backbone. If it needs to be enlarged (or shrunk), the layout lock is acquired in layout change mode and the modification is performed as discussed in Subsection 3.1.2. Finally, the layout lock is released.

Listing 6. DLTree operations

```cpp
void layoutTree::enlargeBackbone()
{ 
  llock.startLayoutChange();
  sortedvector v;
  for each treelet tl in backbone do 
  { 
    tl->acquire();
    v.addItem(tl);
  }
  for each treelet tl in backbone do 
  { 
    tl->release();
  } 
  BuildBalancedTree(v);
  llock.endLayoutChange();
}
```

3.1.4 Assumptions

While the DLTree is always correct, it provides performance benefits assuming some (realistic) behavior of the application. First, although the DLTree balances treelets during layout changes, it does not provide a balancing guarantee. Thus, like any unbalanced tree, it will not work well for workloads that significantly deviate from uniform.

In addition, the DLTree performs best when the tree layout (backbone) is static for the majority of the time. It is less effective when the tree size changes drastically during execution. For example, for a workload of 100% insertions, layout changes would consume a significant part of the execution time.

Of course, if the layout is completely fixed throughout the execution, then there is no need for the layout lock. The layout never changes. However, this extreme stability is not very realistic. Usually, data structures are allocated empty and items are inserted thereafter. In some cases, the stable running range of keys is not even known when the first element is inserted into the data structure. Thus, a design that does not deal with layout changes is impractical.
handled by resurrection. Therefore, frequent merges are triggered when the side buffer over time. In the case of Concurrent Array Set, a merge (i.e. layout change) is triggered when the side buffer performs best in cases where the layout does not change frequently. This operation is illustrated in Figure 3. In the frequent case, the layout lock allows normal operations to treat them to be easily resurrected.

To avoid frequent costly modifications to the sorted array, we apply the following two ideas. First, insertions are applied into a side buffer, which absorbs multiple insert operations before the changes are merged into the sorted array. Occasional merging of the side buffer into the sorted set helps shift most of the search work towards the easily searchable sorted part.

Second, we designed our sorted array to work best with distributions in which elements tend to recur, i.e., there is a good chance that removed elements will be re-inserted to the set again. Thus, in our set we optimistically keep deleted elements (for a limited time) in the sorted array marked as deleted, but without actually removing their entries, allowing them to be easily resurrected.

In order to correctly merge the side buffer into the sorted array, the threads concurrently accessing the array are synchronized by considering the merge operation as a layout change. This operation is illustrated in Figure 3. In the frequent case, the layout lock allows normal operations to treat the sorted array as if it was read-only, thus avoiding complex synchronization. This way, the proposed algorithm is able to keep the initial simplicity of a sorted array.

Similarly to the DLTree, the Concurrent Array Set also performs best in cases where the layout does not change frequently over time. In the case of Concurrent Array Set, a merge (i.e. layout change) is triggered when the side buffer reaches its full capacity. Therefore, frequent merges are avoided when elements are added to the side buffer at a slow rate, which is indeed the case when most inserted objects are handled by resurrection.

4. Measurements

In this section, we study two questions. First, we ask whether layout modifications are worthwhile for the examples of section 3. We then turn to look at concurrent executions with layout changes, and check whether the proposed layout lock provides good performance when compared against available synchronization mechanisms such as StampedLock (Lea and JSR-166), SeqLock (Lameter 2005), and a standard reader-writer lock. Thus, for each of the data structures provided in Section 3 we consider two sets of competitors. First, we consider state-of-the-art designs for concurrent sets. Second, we consider synchronizing the proposed algorithm by one of the available synchronization techniques.

We ran measurements on a 64-cores machine, featuring 4 AMD Opteron(TM) 6376 2.3GHz processors, each with 16 cores. The machine has 128GB RAM, an L1 cache of 16KB per core, an L2 cache of 2MB for every two cores, and an L3 cache of 6MB per half a processor (8 cores). An Ubuntu 14.04 (kernel version 3.16.0) is used. Each execution was repeated 10 times and lasted for 5 seconds. We report the average number of executed operations (throughput).

The concurrent array set was implemented in Java and run on OpenJDK 1.8.0.91 JVM. Results on HotSpot 1.8.0.91 are similar. The DLTree was implemented in C++ since it requires cache-aligned allocation, which is not currently supported by Java. It was compiled using the g++ compiler version 4.9.3 with the -O3 optimization flag.

4.1 Measuring the DLTree

We incorporated the DLTree into the framework of (David et al. 2015), and used their stress-test benchmark. The percentage of search operations (contains) was set to 80%; and insert and remove operations were chosen with a probability of 10% each. We varied the key ranges between 8K, 128K, and 2M, and also measured a key range of 32K to 128K, and 2M, and also measured a key range of 32K to demonstrate an additional interesting behavior. The key to be searched, inserted, or deleted was chosen uniformly at random in the key range. Thus, the tree size was approximately half of the key range. The tree was populated to its approximate size before the measurement started. We varied the number of threads as powers-of-two between 1 and 64.

Since the tree is pre-populated, the benchmark measures the steady-state performance of the DLTree, when there are no layout changes. Yet, the readers and writers execute their layout synchronizations to make sure that they ran safe even if a layout change does occur. We also added a benchmark that measures the build time of a tree with one million elements. This benchmark executes 100% insertions that triggers several layout changes.

For the tree micro-benchmark we consider four state-of-the-art concurrent trees: Bronson et al. (2010) relaxed balanced AVL tree (denoted bronson), Natarajan and Mittal (2014) lock-free tree (denoted natarajan), Drachshler et al. (2014) logical-ordering binary search tree (denoted drach-.
sler), and Ellen et al. (2010) lock-free tree (denoted ellen). For all these trees we took the implementations provided by (David et al. 2015). To the best of our knowledge, these are among the best performing concurrent tree designs. In addition to the DLTree algorithm synchronized by the layout lock (denoted DLTree), we consider the same algorithm of the DLTree when synchronized by a reader-writer lock (denoted DL_RW) and the same algorithm synchronized by SeqLock (denoted DL_SeqL). The SeqLock implementation was taken from the Linux kernel implementation version 4.6. Note that this Linux implementation of the SeqLock does not support concurrent writers, but only optimistic readers and a single layout changer. However, this is sufficient for the DLTree implementation because the backbone (which uses the layout lock) is modified only via a layout change.

**Results** The results for the tree benchmark are presented in Figure 4 as ratios between the throughput of the given implementation and that of Bronson’s tree. Namely, a result of 200% means 2x faster than bronson. The absolute throughput values of the baseline are presented in Figure 4 (e).

The performance of the DLTree compared to the other implementations varies between the different tree sizes. It turns out to be excellent for trees of large size. For trees of size 4K, the DLTree performs similarly to the lock-free tree natarajan, and better than most other competitors. For trees of size 64K, the DLTree is consistently faster than natarajan with 13% improvement for 64 threads. Finally, for large trees of size 1M, the DLTree performs much better than all competitors (35% faster than the bronson, the second-best implementation for this configuration, for 64 threads.)

The DLTree performs better on larger trees due to the fast traversal of the backbone. But we also added trees of size 16K, where the DLTree performs close to Bronson’s tree, but is less efficient than natarajan. Interestingly, the discrepancy between the 4K and the 16K sizes represent a general trend, which is determined by the average treetlet size. The smaller the treetlet, the faster the algorithm, as it spends most of its time in the fast backbone. The average treetlet size is 1 for trees of size 4K, 64K, and 1M, but the average treetlet size is 4 for trees of size 16K. This dependency on the average treetlet size is shown clearly in Figure 5, where we present performance results for 64 threads with varying tree sizes. As can be seen, the DLTree performance is best when its size is a power of 16, where the size of each treetlet is 1 on average, and it deteriorates when this size increases.

Note that we could use a more aggressive heuristic for enlarging the tree, making the backbone larger than the number of elements just to make sure that each treetlet is small on average all the time. This leads to a memory vs. performance trade-off, which we did not pursue further in this paper.

Comparing the DLTree that uses the layout lock implementation to enable layout modifications with a reader-writer lock implementation, the latter is doing quite well for one or two threads, but deteriorates quickly as the number of threads increases. Compared to the SeqLock, it again performs excellently for a few threads (in some cases even slightly better than layout lock), but is 24% – 30% slower for 64 threads. It should be noted, that the SeqLock requires that an optimistic reader starts by reading the version number and storing it in a local field (a register or a stack location). When it finishes, it re-reads the version and compares it to the stored version. Thus, it requires two loads instructions per read operation (the layout lock requires only one) and occupies a register for the duration of the read. In addition, both of these loads limit the compiler’s ability to re-order load instructions, while the layout lock limits it only once. Furthermore, accessing a single location by all threads may be slower than accessing a location per thread as is reported elsewhere (Lin et al. 2015). This may explain the observed performance benefits of the layout lock over SeqLock.

Finally, we created a workload with a 100% insertion rate, which stops immediately after the backbone size reaches a million elements. As discussed in Section 3.1.4, 100% insertions represent a worst case scenario for the DLTree. In addition, a million elements represents a worst case stopping condition, as the workload stops immediately after the entire tree is restructured. That is, the measurement stops immediately after the backbone size is enlarged from $2^{16}$ treetlets to $2^{20}$ treetlets. In this case, the DLTree built the tree in 440 ms while other concurrent trees built such a tree in 180 – 200 ms. The layout changes took approximately 250ms in this case, explaining the performance gap. This demonstrates that layout changes are effective only if a lot of operations are able to benefit from the new layout.

### 4.2 Measuring the Concurrent Array Set

Similar to the DLTree benchmark, the Concurrent Array Set was measured using 80% searches, 10% insertions, and 10% deletions. The Concurrent Array Set works well with uniform workload over a small to medium range (up to 256K) because all the keys in the range are eventually placed in the sorted array (recall that removed values are only logically removed). To avoid a best case scenario test, we also ran the Concurrent Array Set with a workload of a normal distribution (with a theoretically unlimited range).

The Concurrent Array Set was compared against the concurrent skip list set of the standard Java library as well as the Java implementation of Bronson’s tree from (Gramoli 2015). In addition, to demonstrate the benefits of the layout lock in the design of the Concurrent Array Set, we implemented the layout changes synchronization using StampedLock (that is provided by the Java standard library), as follows. Readers use the StampedLock’s optimistic readers interface. Writers use the non-optimistic readers interface since they do not modify the layout but must prevent it from being modified during the write. Finally, layout changers use the writer interface since they modify the layout.

Furthermore, in order to demonstrate the benefit of optimistic reads for synchronized layout changes, we have also
implemented the standard (naive) solution of using a reader-writer lock to synchronize layout changes. That is, a read lock was used for read and write operations, and a write lock was used for layout change operations.

**Results** The results of the Concurrent Array Set benchmark are presented in Figure 6 for normally distributed keys with \( \sigma = 16K \). Compared to Java’s concurrent skip list, the Concurrent Array Set performs 87% faster on a single thread, and 2.7x faster on 64 threads. This demonstrates excellent performance of binary searches over a sorted array, which is enabled by using the layout lock. Results for the uniform distribution with a key range of 256K are presented in Figure 7, and show a similar trend to that of the normal distribution. Results for smaller key ranges (shown in Figure 7(b)) also exhibit similar or stronger relative performance of the Concurrent Array Set, while for large enough sets, the overhead of layout changes becomes too high and performance degrades significantly. Hence, the benefits of the Concurrent Array Set depend on achieving a slow rate of layout changes, which, in turn, depends on the stable running range of keys and the execution time.

Figures 6 and 7 also compare the Concurrent Array Set that uses the layout lock to Concurrent Array Set implementations that use a reader-writer lock (RWLock) or StampedLock for the layout change synchronization. Both latter implementations perform poorly compared to the one using the layout lock when the number of threads is high. Thus, the same performance benefits that we see for the Concurrent Array Set with layout lock are not achieved with current locking mechanisms, even when optimistic locking is applied (in the case of StampedLock) in a manner similar to our proposed paradigm.

**5. Related Works**

Compiler optimizations that generate code that modifies the memory layout are well known (Kennedy and Ziarek 2015; Kandemir et al. 1999a; Mannarswamy 2009; Ding and Kennedy 1999). But they are avoided when parallel accesses are involved. The layout lock may be used to enable such optimizations with concurrent programs as well.
A scheme for internal modifications of data structure layouts was proposed by (Xu 2013). There, the underlying representation of the data structure is modified dynamically. The scheme was able to support data structures synchronized by a global lock, but with no finer-grained synchronization. The layout lock paradigm and implementation extends the reader-writer lock. Mellor-Crummey and Scott (1991) propose a scalable design, using a list of waiting processes, where upon releasing the lock it is passed to the next process in the list. Hsieh and Weihl (1992) proposed a design that is very efficient for infrequent writes, and we used it as the base for the layout lock implementation. Many other implementations exist in the literature, e.g., (Lev et al. 2009; Brandenburg and Anderson 2010; Calciu et al. 2013). A recent paper (Liu et al. 2014) presented a very fast reader-writer lock for the TSO memory model. But this design relies on strong kernel support, not always available.

The layout lock interface bears a resemblance to the SeqLock (Lameter 2005) and StampedLock (Lea and JSR-166), which are synchronization techniques that allow optimistic readers. Both of these techniques were discussed in the introduction. As shown in the Measurements section, the layout lock provides better scalability than both of these methods.

Read-Copy-Update (RCU) is a technique for synchronizing readers and writers that avoids blocking readers. Soules et al. (2003) describe RCU-based layout changes for the K42 kernel. They defined a protocol for the state transfer and exemplify layout changes for various file-system operations. This solution requires kernel support. Tripllett et al. (2010) used a user-space RCU solution to build a dynamically growing hash map. They used RCU primitives to move nodes between buckets while ensuring that a reader of a bucket in the old table would observe the appropriate data. An RCU-based implementation for adjusting the size of a vector (of semaphore pointers) was considered by (Arcangeli et al. 2003).

There are multiple RCU variants for user-level code (Desnouyers et al. 2012). Quiescent-State-Based Reclamation (QSBR) provides the least overhead for reads (practically none) and is widely used for kernel-level implementation. However, user space libraries, such as ones providing concurrent data structures, cannot generally use the QSBR variant, as they cannot identify quiescent states in the entire application. Other RCU-variants include general-purpose RCU and bullet-proof RCU. These variants can be used in user-level libraries, but typically result in high overhead for readers. Hence we did not consider using these.

Recently, it was show that it is possible to avoid the expensive memory fence on readers by infrequently issuing a process-wide memory fence, either by OS support such as the Linux sys_membarrier call (McKenney et al.) or via hardware support (Dice et al. 2016). This RCU-variant can be used on user level libraries and exploring its usage for modifying the layout of data structures is an interesting avenue for future work.

The technique used in this paper, of using a thread local flag for detecting optimistic executions going astray, was first introduce by Cohen and Petrank (2015a,b) in the context of memory management for lock-free data structures.

6. Conclusions

This paper studied concurrent layout modifications as a means to improving performance of concurrent algorithms. We introduced the layout lock paradigm and a layout lock implementation which allows concurrently modifying the layout during the execution with a negligible overhead for read operations and with a small overhead for write operations on the shared data.

We demonstrated the benefits of the layout lock implementation by presenting new algorithms that benefit from layout modifications. Measurements show that performance improvements can be significant. One of these constructions, which is of an independent interest, is a concurrent tree design that performs fast on large trees.

References


