Technical Report

Automatic Fine-grain Locking Insertion for Shared Data Structures

Haiyang Liu Tingting Hu Zongyan Qiu

LMAM and Dept. of Informatics, School of Math, Peking University, Beijing, China
liuhaiyang@pku.edu.cn, hutingting.math@pku.edu.cn, qzy@math.pku.edu.cn

March 31, 2017

Abstract

Applying scalable locking strategies is one of the key challenges in concurrent programming. Although the fine-grain locking gains considerable efficiency improvement in contrast with the coarse-grain techniques, it is tough to program, as well as error-prone. This paper presents an approach, based on program analysis, to automatically add fine-grain locking primitives to produce shared data structures in object-oriented programs. For tree-like structures, the codes produced by our approach are guaranteed to be thread-safe. Experiments show that the approach can successfully deal with programs which are challenging to handle manually. In addition, the entire analysis in our approach is static, thus is very efficient.
Keywords: Concurrency, Fine-grain Locking, Serializability, Shared Data Structures, Program Analysis

1 Introduction

The blooming of multi-core processors leads to the urgency for the multithreaded software. However, it is well known that concurrent programs are notoriously hard to develop. To relieve the burden, programmers tend to use coarse-grain high-level synchronization facilities, such as `synchronized` method in Java [8].

Data race [22] is one of the most important failures in multithreaded programs. A data race occurs when two or more operations from different threads may access the same memory location, while at least one of the operations is a write, and there is no synchronization to mandate exclusiveness among them. When a data race occurs, the behavior of the program relies on the speed of threads and the interleaving of their operations. Suppose there is a class which is intended to be used by multiple threads. Programmers usually define the methods of the class as `synchronized` to prevent data races.

However, the coarse-grain locking may result, in fact, that many operations are executed sequentially, thus sacrifices too much parallelism [19, 3, 7]. The fine-grain synchronization mechanisms, e.g. locks and semaphores, may be used to get better performance. For example, we know that multiple threads are allowed to read a shared object, when there is no write operation. The dark

side is, the lower-level primitives are very hard to handle and error-prone [20]. It is urgent to develop strategies and techniques to get a good trade off between the easy-to-use and the performance.

To overcome the dilemma, we present here an automatic technique which can transform a class definition without synchronization controls to a shareable one by adding proper lock operations. Our transformation guarantees data race freedom for the resulting classes. Furthermore, we prove that the locked range is minimum, thus permits a high degree of parallelism. In addition, for the tree-like data structures, the resulting classes are dead-lock free. Some experiments show that the fine-grain synchronization produced by our approach results in good performance and scales well. Technically, our method analyzes every method of the class, find out the start and end points where each object is accessed, then insert lock/unlock instructions at the correct positions.

Here is a summary of our contributions:

- We develop a technique to automatically insert locking instructions into classes which are intended to be shared by multiple threads.
- We prove that our approach guarantees data race freedom. In addition, for the tree shape data structures, our technique guarantees the thread safety, and the minimum locked range for the shared objects.
- We conduct some experiments which indicate the scalability and performance of our approach.

The rest of this paper is organized as follows. Section 2 illustrates the ideas behind our approach using some examples. Section 3 introduces some fundamental concepts. Section 4 describes our approach and algorithms in details. Section 5 proves the properties for our approach. Section 6 presents our implementation and gives some experimental results. Section 7 discusses the related work. Finally, Section 8 concludes and discusses some future work.

2 Examples

We give two simple examples in this section to illustrate our idea.

The first example is a singly linked list class `LList` and a list node class `Node`. The lists are assumed to be used by multiple threads.

Taking the synchronized idioms in Java providing coarse-grain synchronization, we can implement the class as a monitor, as shown in Figure 1. Method `size` (and some others) of the class is declared as `synchronized` to prevent race conditions. In the execution, a `synchronized` method automatically performs a lock action when invoked, and its body is not executed until the thread holds the lock. When a `synchronized` method terminates, a corresponding unlock action is automatically performed [8]. Thus, there are implicit acquire/release operations at the entry/exit of method `size` and other methods, respectively. Clearly, when a thread is executing a synchronized method, e.g. `size`, the coarse-grain locking mechanism prevents other threads to execute any synchronized method of the list, and the result is the relatively low performance.

The fine-grain locking allows higher parallelism. There are a number of objects involved in the execution of `size`, including `this` object (the list object) and the objects referenced by variable `n` in the execution (i.e. the nodes). We

```
class Node {
    Node next;
    T data;
}
```

```
class LList {
    Node head;
    synchronized int size() {
        int sz = 0;
        for (Node n = head; n != null; n = n.next)
            sz++;
        return sz;
    }
    synchronized void insertFront(T data) { ... }
    ...
}
```

Figure 1: `LList` class with synchronized methods

```
int size() {
    int sz = 0;
    acquire(this);
    Node n = head;
    acquire(head);
    release(this);
    while (n != null) {
        sz++;
        Node n1 = n.next;
        acquire(n.next);
        release(n);
        n = n1;
    }
    release(n);
    return sz;
}
```

Figure 2: Fine-grain locking for method `size`

associate each object with a lock to get fine-grain locking. Since each access to the objects affects only a part of the execution of `size`, we can perform the locking locally for the affected operations to increase parallelism. For instance, the `LList` object `this` is accessed only once in the method body: `this.head` field is read at the beginning of the method. Thus we can lock object `this` around the statement to prevent data race. Furthermore, object `head` would be accessed after it is assigned to `n`. We need to ensure that `n` cannot be changed by other threads until the last operation to `head`. This requires to use a "hand-over-hand" locking, so that `head` should also be locked before `this` is unlocked.

Suppose that statement `acquire(o)`/`release(o)` acquires/releases the lock of `o`. We can get a fine-grain locking version of `size`, as shown in Figure 2.

In fact, we can automatically produce the same result for method `size`. Using program analysis techniques, we can trace each access operation of each shared object, calculate the range of program protected by locks for each object, and insert the locking statements at proper positions. The analysis for loops is a bit complicated. We can see in this example, the object referenced by variable `n` changes in each iteration, thus we need to acquire and release the lock in each

```
class CachedData {
    Object data;
    boolean cacheValid;

    synchronized void
    processCachedData() {
        if (!cacheValid) {
            data = getData();
            cacheValid = true;
        }
        use(data);
    }
}
```

```
void processCachedData() {
    acquireWrite(this);
    acquireRead(this);
    if (!cacheValid) {
        data = getData();
        cacheValid = true;
        releaseWrite(this);
    } else {
        releaseWrite(this);
    }
    use(data);
    releaseRead(this);
}
```

Figure 3: `CachedData` class: (left) with synchronized method, and (right) with automatic readers-writer locking

iteration. In addition, we have to rewrite the loop from the for-statement to while-statement, since read access to variable `n` exists in the update statement of `for` loop. We need to move the `for` update statement inside of the loop and introduce a temporary variable in order to make a correct position for the lock operations. These transformations need some non-trivial analyses to the programs.

Our analysis diminish the range of programs protected by locks to increase parallelism in the previous example. However, this does not always work well. The `CachedData` class in Figure 3 includes both read and write operations. Using the same techniques as before, a lock will be inserted at the beginning of `processCachedData` method, and an unlock will be inserted at the end of the method. This gives the same coarse-grain protection of `synchronized` method, without utilizing the difference features of the read/write operations.

We can improve the result by separating the analyses of read access and write access and using *readers-writer locks* (i.e. *shared exclusive locks*) to achieve synchronization. For an object `o`, we assume that statements `acquireRead(o)`/`acquireWrite(o)` (`releaseRead(o)`/`releaseWrite(o)`) acquires (releases) the read lock or write lock of `o`, respectively. Then, our approach can produce the version of `processCachedData` as shown in Figure 3 (right).

In `processCachedData`, the analysis about object `this` involves conditional branches which should be handled separately. There is one read operation to object `this` in the condition, therefore, the read-lock should be acquired before this read. There are two write operations to object `this` in the true branch, and no operation in the empty else branch. To prevent deadlocks, the write-lock should be acquired before acquiring the read-lock. But the write-lock should be released after the last write operation (the statement `cachedValid = true`) in the true branch, and at the beginning of the else branch. After using `this.data`, the read-lock can be released. In this way, we achieve accurate and compact locking in both branches.

This adjustment is also done automatically in our approach. Even for the complicated read/write situations, it can still insert locks in correct positions.

3 Fundamentals

Before discussing our automatic locking positioning algorithm, we give an overview and introduce some fundamental concepts related to our work.

3.1 Goals and Assumptions

Our main aim is to ensure the thread-safety of shared data structures (objects), and to develop a method to make this automatically.

Having a data structure with a set of operations for sequential programs, we are going to transform it to a concurrent data structure which is thread safe. We assume that the data structure, as well as its methods, is described as a class in an object oriented language originally, e.g., in Java. After our transformation, the methods of the data structure may be safely invoked in the multi-threaded programs. To simplify the work, we suppose that the methods are self-contained, and they never invoke each other.

3.2 Locking States of Objects

The data race is caused by concurrent access to the shared objects. In object-oriented languages, accessing an object means reading or writing some fields of the object. Thus, we are going to prevent data races by using locks to restrict the concurrent accesses to the shared objects.

However, locking every single read/write instruction is too fine-grain. In addition, although this ensures data race freedom, it may not ensure an appropriate concurrent semantics as expected. If an object is accessed several times successively by a thread, those read/write operations might be related. For example,

```
o.n = o.n + 1;
```

The statement accesses `o` two times. To guarantee the correct semantics, the two successive read and write to object `o` should not be interrupted by other threads. However, the strategy to lock each read/write operation individually cannot ensure this protection. Similarly, assume following code segment

```
o.n = 5;
... // some statements that do not access o
x = o.n;
```

appears in a method. In the sequential execution, the last statement will assign 5 to `x`, since `o` keeps unchanged between the two statements. In the multiple threads circumstance, lock each read/write operation individually cannot keep the semantics. The problem is very practical. The semantic restriction requires that a thread to acquire the lock before the first access to an object, and release the lock after the last access in an operation unit.

To address above problem, people propose the 2PL locking protocol [6] as a solution. To follow the protocol, a thread should acquire all the locks related to the objects used in an operation at the beginning of the operation, and release these locks at the end. This strategy ensures the atomicity. However, the down side of 2PL is that it is often too coarse grained when applying to many data structures. For example, to implement a concurrent list (or a tree), the 2PL strategy asks to lock the entire list (or tree) during each operation, no matter

how long it takes to finish the operation. In this case, when one thread *th*, for example, traverses a list (or tree), other threads, which want to access the same structure, must wait until *th* releases it. This implies that, under 2PL, the shared data structures can only be accessed alternatively (and sequentially) by the threads. This is really not ideal.

To increase concurrency, we use the object based locking protection strategy. We associate a lock to each object. Locking an object makes no restriction on the access to another object when the thread holds it. Then we use a hand-over-hand locking strategy to protect the data structure. With this locking scheme, we can relax the limitation of 2PL and support more concurrent accesses. However, although this strategy makes the great potential, it will lie also heavy burden on the programmers. In developing a data structure, the programmers must try to determine when an object should be locked or released, and insert suitable lock/release primitives into their code. Clearly, this work is really error-prone.

Fortunately, this dilemma is not unsolvable. We have developed a method (an algorithm with implementation) to analyze the access range of each object in methods, which can determine the precise position to acquire and release locks. Our algorithm traces the access of each object, and obtain the correct locking range finally. Informally, given an object `o`, for each statement *s* in a method, we determine whether the statement is in the access range of `o`, that is called the *locking state* of `o` at *s*. When a thread runs from a non-locking state into a locking state of `o`, it needs to acquire the lock of `o`; and when it runs out of a locking state of `o`, it should release the lock of `o`.

3.3 Reentrant Readers-Writer Locks

To increase concurrency further, we are going to use the reentrant readers-writer locks, as shown in the example given in Figure 3.

A *readers-writer lock strategy* maintains a pair of associated locks with each shared object, one for read-only operations and one for writing. The read lock may be obtained simultaneously by multiple reading threads, so long as there is no a writer. Correspondingly, the write lock is exclusive, which can be obtained by at most one thread. When a thread *th* gets the right to write the object, all other threads, no matter they want to write or read the object, will be blocked until *th* finishes its work and releases the write lock.

Furthermore, we assume that locks used in our work are all *reentrant*. This means that a thread can relock a lock which it holds currently, without causing a deadlock. In addition, a writer thread (which holds a write lock of an object) can reacquire the corresponding read lock, but not vice-versa.

4 Automatic Locking Generation

In this section, we present our approach in details, specially our analysis algorithm for automatic lock generation (insertion of lock acquire/release operations). The procedure consists of the following steps:

1. Transform the source code into Static Single Assignment (SSA) form [1] in the 3-address representation and calculate the points-to relationships.

2. Calculate the set of locks which are necessary to hold for each program location to prevent data race and the interruption of operations, using intra-procedure dataflow analyses.

3. Expand the lock sets to ensure further properties of thread safety.

4. Insert locking statements in correct positions according to the lock sets.

In the following subsections, we give details for each of these steps.

4.1 Analysis of References

In the first step, we need to figure out reference relations between existing objects, and points-to relations between the variables and objects. These relations can be very complicated. On the one hand, throughout the method to be analysis, one variable/field may reference to different objects. On the other hand, there may be several variables/fields referring to the same object.

Some basic cases exist which can make one variable to refer to different objects in a method. First, a variable may be reassigned to another object in the sequential execution of statements. A special case is that a variable may be used in a loop to iterating over a series of objects, for instance, in the traversal loop for a linked list:

```
while (x.next != null)
    x = x.next;
```

Therefore, we need to determine the variables which *may* point to different objects during the execution. We use SSA form to deal with these cases. If a variable is reassigned, possibly in difference control branches, we split it into different variables. And the ϕ-function in the SSA form makes it clear that a variable may hold different values through different execution paths. Here is the SSA form of above `while`-loop after the splitting:

```
L0:
  x1 = phi(x, x2);
  t = x1.next;
  b = (t == null);
  if (b) goto L1;
  x2 = x1.next;
  goto L0;
L1:
  x3 = phi(x1, x2);
```

Here variable `x` has been split into `x1`, `x2` and `x3`, which may point to different objects. We can then analyze the locking states for each of those objects.

To prevent deadlocks, our later lock set expanding algorithm (Figure 7) needs to know the point-to relations. We identify the cases that different variables may point to the same object using context-insensitive point-to analysis. Suppose *alias* is a points-to analysis, we assume that *alias*(P), where P is a segment of code, is a function α_P which maps any pair of variables and/or fields to a boolean value, to tell whether the two may alias with each other or not. We call *alias* sound, if for any program P and any two variables and/or fields a, b in P, a and b never point to the same object in any execution of P when $\alpha_P(a, b) =$ false. And we call *alias* accurate, if $\alpha_P(a, b) =$ true implies that a

and b alias with each other. For static analysis, a sound and accurate points-to analysis is impossible [16, 21]. A practical static points-to analysis (as the one we used) is guaranteed to be sound, but not always accurate. In the following sections, we will suppose using a sound points-to analysis.

Nevertheless, after the first step, we obtain a CFG (Control Flow Graph) of the processed method in the SSA-form. This is the input of the next step.

4.2 Analysis of the Read and Write Lock Sets

In the second step, we use dataflow analysis algorithms (ref. [1]) to track the set of locks (read or write locks) that are necessary to protect the shared objects. Generally speaking, our algorithm calculates the minimum sets of locks that must be acquired at each program location to prevent data race and interruption of operations. Its input is the CFG of a method in the SSA-form obtained from previous step. Its output is the lock sets for each program location.

We introduce some concepts:

Definition 1. *We say object o* was accessed *at location s in execution path p, if there is some access of o from the entry point of the method to s in p. And we say o* will be accessed *at s in p, if there is access of o from s to the exit point in p.*

We define that the May-Before *lock set at program location s is the union of each lock of object, which was accessed at s in some path p up to s. And the* Must-After *lock set at program location s is the intersection of the locks of the objects, which will be accessed at s in every path p through s.*

Our algorithm performs a two-stage intra-procedure dataflow analysis. In the first stage, it uses a forward analysis to decide the May-Before lock sets. In the second stage, it conducts a backward analysis to decide the Must-After lock sets. We will prove in section 5, that the intersection of these two sets, at each program location, is the minimum set of read/write locks which are necessary to hold for preventing data race and interruption of the successive operations.

4.2.1 Forward analysis.

The forward data-flow analysis starts from entry point of the method body with an empty lock set.

Given a statement s, let $s.in$ and $s.out$ denote the May-Before sets at the locations before and after s, respectively. Based on all the immediate predecessor t of s, we formulate the propagating of the lock-sets after these t's (i.e. $t.out$) to s (to form $s.in$) as follows:

$$s.in := \bigcup_{t \in prec(s)} t.out$$

For each statement s, to form the May-Before set at the location after s, $s.out$, we take the union of the set $s.in$ and the generated-set of the statement, $s.gen$, then subtract the killed-set of the statement, $s.kill$. That is,

$$s.out := (s.in \cup s.gen) \setminus s.kill$$

Table 1: Terms in the Transfer Function

Statement	*s.gen*	*s.kill*
`o.f = x`	{`o.writeLock`, `o.readLock`}	∅
`x = o.f`	{`o.readLock`}	∅
Otherwise	∅	∅

```
function ForwardAnalysis(CFG)
    lock ← ∅
    worklist ← queue()
    for stmt ∈ CFG.nodes do
        lock[stmt] ← ∅
    worklist.enqueue(CFG.nodes)
    while worklist is not empty do
        stmt ← worklist.dequeue()
        old ← lock[stmt]
        in ← ⋃_{n∈CFG.prec[stmt]} lock[n]

        new ← Transfer(stmt, in)
        if old ≠ new then
            worklist.enqueue(CFG.succ[stmt])
            lock[stmt] ← new
    return lock
```

```
function Transfer(stmt, in)
    if stmt is o.f = x then
        gen ← {o.writeLock, o.readLock}
    else if stmt is x = o.f then
        gen ← {o.readLock}
    else
        gen ← ∅
    kill ← ∅
    return (in ∪ gen) \ kill
```

Figure 4: Forward Analysis Algorithm

This transfer function takes the standard form to calculate how one statement affects the lock sets. To complete the definition, we should define the *gen* and *kill* sets for each statement form.

To prevent data race, before each access on object `o`, i.e. a reading or writing action on `o.f`, the thread needs to hold the read or read/write locks of `o`. As pointed in section 4.1, since a variable is assigned only once in the SSA form (except in the loop iterations), the variable names in the SSA-formed CFG can be thought as the indicators of objects in the analysis. Then we take the definitions for the *gen* and *kill* given in Table 1.

This propagating process iterates until the sets at all the program locations reach a fixed point. Figure 4 gives the forward analysis algorithm.

4.2.2 Backward analysis.

The backward dataflow analysis works similarly as the forward analysis, except the direction of the dataflow propagation and the merging operation. This analysis stage calculates the Must-After lock set for each program location. For statement s, based on the immediate successors t's of s, the lock set at the location after s, $s.b\_out$, is the intersection of the lock sets of the locations before t ($t.b\_in$), that is

$$s.b\_out := \bigcap_{t \in succ(s)} t.b\_in$$

```
function BackwardAnalysis(CFG)
    lock ← ∅
    worklist ← queue()
    for stmt ∈ CFG.nodes do
        lock[stmt] ← ∅
    worklist.enqueue(CFG.nodes)
    while worklist is not empty do
        stmt ← worklist.dequeue()
        old ← lock[stmt]
        out ← ⋂_{n∈CFG.succ[stmt]} lock[n]
        new ← Transfer(stmt, out)
        if old ≠ new then
            worklist.enqueue(CFG.prec[stmt])
            lock[stmt] ← new
    return lock
```

Figure 5: Backward Analysis Algorithm

```
function ComputeLockSets(CFG)
    lock_fwd ← ForwardAnalysis(CFG)
    lock_back ← BackwardAnalysis(CFG)
    w_set0, r_set0 ← ∅, ∅
    for node ∈ CFG.nodes do
        w_set0[node] ← lock_fwd[node].writeLocks ∩
lock_back[node].writeLocks
        r_set0[node] ← lock_fwd[node].readLocks ∩ lock_back[node].readLocks
    return w_set0, r_set0
```

Figure 6: Compute the Lock Sets by Intersection

The Must-After lock set at the location before s is obtained with the similar propagating function as the forward analysis, but reversed in direction,

$$s.b\_in := (s.b\_out \cup s.gen) \setminus s.kill$$

The generated-set and killed-set of statements have the same definitions as the forward analysis, as given in Table 1. The algorithm makes the propagation repeatedly until all the sets reach a fixed point (Figure 5).

4.2.3 Locking Rules

After the forward and backward analyses, each program location in the analyzed CFG is associated with two lock sets. We take the intersection of the two out sets of each statement as the lock set of the statement (Figure 6). A read (write) operation of `o` is permitted to performed at a statement only if the read (write) lock of `o` is in the lock set of the statement. Then we can define the rules to insert the locking statements. Note that each statement is represented by a node in the CFG.

The basic rules are intuitive. If an object is locked at one node in the CFG but not locked at one of its subsequent nodes, then a release statement should

Table 2: Locking Operation According to the Transitions of Lock Sets

Before\After	∅	ReadLock	Read+WriteLock
∅	—	acquireRead	acquireWrite
ReadLock	releaseRead	—	*Not Allowed*
Read+WriteLock	releaseWrite	acquireRead; releaseWrite	—

be inserted between these two nodes; on the other hand, if an object is unlocked in one node but locked in one of its subsequent nodes, then an acquire statement should be inserted between these two nodes.

However, for readers-writer locks, the rules are a bit more complicated. Due to the algorithm TRANSFER (Figure 4), a write lock of object `o` is always associated with a read lock of `o` in the lock set, which is used to indicate that the write operation is exclusive from other concurrent read or write operations to the same object. Because the write locks exclude both concurrent reads and writes, we insert only acquireWrite/releaseWrite for the transition between empty and Read+WriteLock for an object. When a downgrading from a write lock to a read lock is required, we acquire the read lock first, then release the write lock. Upgrading from the read lock to a write lock is not allowed, because it may introduce deadlock.

The final rules are given in Table 2.

4.3 Lock Sets Expansion

However, there are still problems to be solved before the insertions of the locking statements:

1. To prevent other threads from destroying the data structure and to keep serializability, a thread may acquire the lock of the object referenced by field `o.f` only if it is holding the lock of `o`. In addition, if we need to acquire the write lock of `o.f`, we must hold the write lock of `o`.

2. Although downgrading from the write lock to a read lock is safe, upgrading from a read lock to the write lock may introduce deadlocks in concurrent executions, thus must be forbidden.

3. In the previous dataflow analysis stages, objects are indicated as SSA formed variable names, which may be still aliases at runtime. The locking ranges of alias references should be merged to prevent the object being locked and released multiple times, to prevent that other threads interfere in between to make destructions.

All these problems can be solved by expanding the lock sets. To protect the data structure, we need to expand the locking range of field `o.f` backward until the locking ranges of `o` and `o.f` are overlapped. To eliminate deadlocks from upgrading of locks, we need to expand the range of write locking backward to where we acquire the corresponding read lock, if it exists. The locking ranges of alias references v and v' may be either overlapped or not. If they do not overlap, we expand the range of the later locking backward until the two ranges connect,

```
function ExpandLockSets(CFG, α_P, w_set, r_set)
    repeat
        q ← [CFG.exit]
        while q ≠ [] do                                   ▷ Breadth-First Traversal
            node ← q.pop()
            for prev ∈ prec(node), t ∈ prec*(prev) do
                for ∀u, v, f · α_P(v, u.f) do
                    if u.readLock ∈ r_set[t] ∧ v.readLock ∉ r_set[prev] ∧
v.readLock ∈ r_set[node] then                             ▷ Condition 1
                        r_set[prev] ← r_set[prev] ∪ v.readLock
                    if u.writeLock ∈ w_set[t] ∧ v.writeLock ∉ w_set[prev] ∧
v.writeLock ∈ w_set[node] then                            ▷ Condition 1
                        w_set[prev] ← w_set[prev] ∪ v.writeLock
                for ∀v, v' · α_P(v, v') do
                    if v.readLock ∈ r_set[t] ∧ v.readLock ∉ r_set[prev] ∧
v'.readLock ∈ r_set[node] then                            ▷ Condition 2
                        r_set[prev] ← r_set[prev] ∪ v'.readLock
                    if v.readLock ∈ r_set[prev] ∧ v.writeLock ∉ w_set[prev] ∧
v'.writeLock ∈ w_set[node] then                           ▷ Condition 3
                        w_set[prev] ← w_set[prev] ∪ v'.writeLock
                q.append(prev)
    until w_set, r_set reach the fixed points
    return w_set, r_set
```

Figure 7: Lock Sets Expansion Algorithm

to prevents the possible interference to the object from other threads. On the other hand, if the two ranges are overlapped, the object's lock may be acquired twice and then released twice, which will not cause any problem because we use reentrant locks.

ExpandLockSets uses an iterated breadth-first traversal to perform locking range expansion (Figure 7). To resolve the potential alias problem, the result of a points-to analysis α_P is utilized. For each statement *node* in the CFG:

- If $\alpha_P(v, u.f)$ and the read/write lock of field v is in the lock set of *node* but not in the lock set of some predecessor *prev* of *node*, and the read/write lock of u is in the lock set of some ancestor t of *prev* (includes), then the read/write lock of v should be also in the lock set of *prev* to connect the locking ranges to protect the data structure (Condition 1).

- If $\alpha_P(v, v')$ and the read lock of v' is in the lock set of a *node* but not in the lock set of its predecessor *prev* of *node*, and the lock of v is in the lock set of some ancestor t of *prev*, which indicates the possibly separated locking ranges, then the lock of v' should be also in the lock set of *prev* to connect the two locking ranges (Condition 2).

- If $\alpha_P(v, v')$ and the write lock of an object v' is in the lock set of *node*, and only the read lock of v is in the lock set of some predecessor *prev*

```
procedure InsertLocking(CFG, w_set, r_set)
    for edge ∈ CFG.edges do
        for lock ∈ (r_set[edge.src] \ r_set[edge.dest]) do
            Add acquireRead(lock.obj) at edge
        for lock ∈ (w_set[edge.src] \ w_set[edge.dest]) do
            Add acquireWrite(lock.obj) at edge
        for lock ∈ (w_set[edge.dest] \ w_set[edge.src]) do
            if lock.obj ∈ r_set[edge.dest] then                ▷ Downgrading
                Add acquireRead(lock.obj) at edge
            Add releaseWrite(lock.obj) at edge
        for lock ∈ (r_set[edge.dest] \ r_set[edge.src]) do
            Add releaseRead(lock.obj) at edge
```

Figure 8: Locking Insertion Algorithm

```
procedure MethodTransform(CFG, α_P)
    w_set0, r_set0 ← ComputeLockSets(CFG)
    w_set, r_set ← ExpandLockSets(CFG, α_P, w_set0, r_set0)
    InsertLocking(CFG, w_set, r_set)
```

Figure 9: Transform the Method

of *node*, then the write lock of v' should also in the lock set of *prev*, to prevent lock upgrading (Condition 3).

Note that in dealing with Condition 2, we expand only the read locks. However, the problematic write locks can be also handled in Condition 3.

Although the intention of the algorithm in Figure 7 is to extend the locking ranges of some objects, it turns out to be represented as expanding the lock sets associated with some program locations in the CFG. After this work, we can obtain the final lock sets for each of the locations in the CFG (of the method), and are ready to insert the necessary lock acquiring and releasing statements into the program (the CFG). Since the objects are indicated as variable names, the locking statements can be generated straightforward.

4.4 The Overall Algorithm

After the locking range expansion, the locking statements are inserted by InsertLocking (Figure 8), using the rules in Table 2,

Then we can describe the overall algorithm which transforms an sequential method to a concurrent one. The algorithm is given in Figure 9, where *CFG* is the CFG of the method, and α_P is the result of alias analysis.

5 Safety and Correctness

Having developed the algorithm, we prove now that it makes a correct transformation for programs. We will first prove that the data structures will be

protected from data race and incidental interference, then the generated locking range is minimum providing that an ideal (accurate) points-to analysis is used. Finally, we prove that for the tree-shaped data structures, our technique ensures serializability and deadlock freedom.

To simplify the discussion, we take some notations from our algorithm. We use $r\_set[s]$ and $w\_set[s]$ to denote the read and write lock sets at statement s, respectively. Similarly, we use $lock\_fwd[s]$ and $lock\_back[s]$ to denote the out lock sets at s given by the forward and backward analyses, respectively.

Lemma 1. *Suppose* $CFG' = \textsc{InsertLocking}(CFG, w\_set, r\_set)$*, where* w_set *and* r_set *are obtained by algorithm* ExpandLockSets*. When a thread th executes this modified* CFG' *up to statement s, then th holds the read lock of object* `o` *at s, if and only if*

$$\texttt{o.readLock} \in r\_set[s], \qquad \texttt{o.writeLock} \notin w\_set[s].$$

And th holds the write lock of object `o` *at s, if and only if*

$$\texttt{o.readLock} \in r\_set[s], \qquad \texttt{o.writeLock} \in w\_set[s].$$

Proof. We consider the write lock first.

On the one hand, suppose that $\texttt{o.readLock} \in r\_set[s]$ and $\texttt{o.writeLock} \in w\_set[s]$, we are going to prove that *th* holds the write lock at s.

Before performing ForwardAnalysis and BackwardAnalysis, $lock\_fwd$ and $lock\_back$ are empty at the start and end nodes of the CFG, respectively, and they are unchanged during the analyses. Therefore, r_set and w_set are also empty at the start and end nodes of the CFG after the dataflow algorithms. Since the CFG is connected, for the execution path p from the start node to s, because $\texttt{o.writeLock} \in w\_set[s]$, there must be a node t between the start node and s such that $\texttt{o.writeLock} \notin w\_set[t]$, and for each node t' between t and s we have $\texttt{o.writeLock} \in w\_set[t']$. We know that $\texttt{o.readLock} \in w\_set[t']$ because of Transfer and the monotonicity ExpandLockSets. Then we know that $\texttt{o.readLock} \notin w\_set[t]$ because of the Condition 2 of ExpandLockSets (otherwise w_set[t] is extended to include `o.readWrite`). Therefore, there should be an `acquireWrite(o)` after t in p inserted by InsertLocking, and there is no `releaseWrite(o)` after that in path p. This implies that *th* holds the write lock of `o`.

On the other hand, suppose that *th* holds the write lock of `o`, we prove that $\texttt{o.readLock} \in r\_set[s]$ and $\texttt{o.writeLock} \in w\_set[s]$.

Since *th* holds the write lock of `o` at s, we know that for any path p up to s, there must be an `acquireWrite(o)` after some statement t in p, and no `releaseWrite(o)` between t and s. By InsertLocking we know that $\texttt{o.writeLock} \notin w\_set[t]$ and $\texttt{o.writeLock} \in w\_set[t']$, where t' is the successor of t in path p. Since there is no `releaseWrite(o)` between t' and s, it's easy to prove (by induction) that for each statement t'' between t' and s, $\texttt{o.writeLock} \in w\_set[t'']$. And then $\texttt{o.writeLock} \in w\_set[s]$. By the algorithm, we know there must also be $\texttt{o.readLock} \in r\_set[s]$.

This completes the proof for the write locks. The assertion about the read locks can be proved similarly. □

In fact, our algorithm ensures that, for any object o, any execution path p, the locking states of o in p take the form as shown in Figure 10. It is not locked

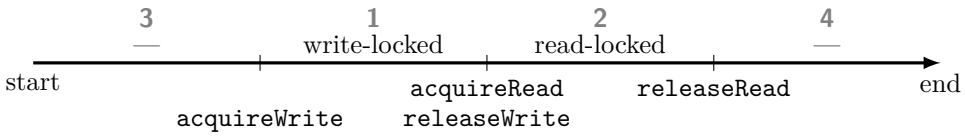

Figure 10: The locking state transitions for an object in an execution path

at the start of p, then is write-locked, then is read-locked, and finally unlocked up to the end of p. The range(s) of the write-locked and/or read-locked may be empty. The locking statements shown in the picture make the transition of the locking states. The numbers marked above are the labels of the ranges used in Proposition 1 below. We have the following proposition.

Proposition 1. *For any statement s in a method m of a (modified) class produced by our algorithm, when a (any) thread th executes up to s, and for any object o, one of the following situations occurs:*

1. *Thread th holds the write lock of o, or*
2. *Thread th holds the read lock of o, and there is no write operation performed by th on o from s to the end of method m, or*
3. *There is no read or write operation performed by th on o from the start of m up to s inclusively, or*
4. *There will be no read or write operation performed by th on o from s to the end of m.*

Proof. By Lemma 1, if $\texttt{o.writeLock} \in w\_set[s]$, then th holds the write lock of o at s. This is situation 1.

Now, let's consider the situation that $\texttt{o.writeLock} \notin w\_set[s]$, but $\texttt{o.readLock} \in r\_set[s]$. By Lemma 1, th holds the read lock of o at s. We prove that there is no write operation performed by th on o from s to the end of method m, that is situation 2.

We prove this by contradiction. If there were a write statement t for $\texttt{o}$ in the execution path of thread th from s to the end of m, then $\texttt{o.writeLock}$ would be always in the output of TRANSFER(t, in). Thus $\texttt{o.writeLock} \in lock\_fwd[t] \cap lock\_back[t]$ according to the exit condition of the loop in FORWARDANALYSIS and BACKWARDANALYSIS. Because w_set is monotonic increased in EXPANDLOCKSETS, we have

$$\texttt{o.writeLock} \in lock\_fwd[s].writeLocks \cap lock\_back[t].writeLocks \subseteq w\_set[t]$$

But by the Condition 2 of EXPANDLOCKSETS, since $\texttt{o.readLock} \in w\_set[s]$, this implies that $\texttt{o.writeLocks} \in w\_set[s]$. This is a contradiction. Therefore, there is no write operation performed by th on $\texttt{o}$ from s to the end of m.

Now, we consider the situations that $\texttt{o.readLock} \notin r\_set[s]$.

Since r_set is monotonic increased in EXPANDLOCKSETS, we have

$$lock\_fwd[s].readLocks \cap lock\_back[s].readLocks \subseteq r\_set[s]$$

this implies that $\texttt{o.readLock} \notin lock\_fwd[s].readLocks$ or $\texttt{o.readLock} \notin lock\_back[s].readLocks$.

Suppose that `o.readLock` $\notin$ *lock_fwd*[*s*].*readLocks*, then we prove that it is situation 3 by contradiction: If there were a write operation t for `o` in the execution path of thread *th* from the start of m up to s, then `o.readLock` would be always in the output of TRANSFER(t, *in*). Thus, we have `o.readLock` $\in$ *lock_fwd*[t] according to the exit condition of the loop in algorithm FORWARD-ANALYSIS, which is a contradiction.

Similarly, if `o.readLock` $\notin$ *lock_back*[*s*].*readLocks*, we can prove that it is situation 4. This concludes the proof for the proposition. □

Corollary 1. *The program generated by* METHODTRANSFORM *is data race free, any object access is protected by locks.* □

Corollary 2. *The program generated by* METHODTRANSFORM *is free of lock upgrading.* □

Besides data race freedom, Proposition 1 says also that an object o accessed in method m is locked and released only once (regardless lock reentrance and the downgrading). Therefore, if there are multiple read/write operations of o, they are always performed within the locking range of o, and can never be interfered by other threads.

Further, by EXPANDLOCKSETS it's easy to see that objects are locked hand-over-hand.

Proposition 2. *If a thread th is executing our modified program, for any object o and its field o.f:*

- *When th acquires the read lock of o.f, it must have held the read or write lock of o, if any.*
- *When th acquires the write lock of o.f, it must have held the write lock of o, if any.* □

We can prove that, for each object `o`, the locking set generated by our algorithm is minimum, which implies minimum locking range.

Proposition 3. *Having the result of an accurate points-to analysis, the locking set (r_set and w_set) generated by* METHODTRANSFORM *is minimum that satisfies the properties described in Proposition 1 and Proposition 2.*

Proof. We prove for the read locks only. Write locks can be proved similarly.

For a CFG of method m, suppose that *r_set* is the locking set assignment generated by METHODTRANSFORM for all the nodes of the CFG. Now we prove that for any locking assignment *r_set*$'$ for the CFG, which satisfies properties described in Proposition 1 and 2, *r_set*[*s*] $\subseteq$ *r_set*$'$[*s*] for each statement s in CFG. Then *r_set* is minimum.

Since the points-to analysis is accurate, we could merge the different variables which are aliases before the dataflow analysis. Then the EXPANDLOCKSETS never merges non-overlapped locking ranges in Condition 2.

Suppose that `o.readLock` $= l \in$ *r_set*[*s*]. Consider the case that

$$l \in \textit{r\_set0}[s] = \textit{lock\_fwd}[s].\textit{readLocks} \cap \textit{lock\_back}[s].\textit{readLocks},$$

which is equivalent to

$$l \in \textit{lock\_fwd}[s].\textit{readLocks} \qquad \text{and} \qquad l \in \textit{lock\_back}[s].\textit{readLocks} \qquad (*)$$

We consider the first assertion of $(*)$. In the last iteration of ForwardAnalysis, we have

$$l \in in \cup gen = lock\_fwd[s].readLocks$$

If $l \in gen$, the current statement is in form of `x = o.f` by the definition of Transfer. Since the points-to analysis is accurate, ExpandLockSets never combines lock sets of two objects. Therefore, we can determine the variable or expression `o` accurately. Thus `o.readLock` $\in r\_set'[s]$ by the definition of safe lock set.

On the other hand, if $l \in in = \bigcup_{n \in CFG.prec[s]} lock\_fwd[n]$, then there is an immediate predecessor n of s such that $l \in lock\_fwd[n]$. We know that either object o is read at n, or there is an immediate predecessor n' of n such that $l \in lock\_fwd[n']$. Therefore, we know by induction that there is a transitive predecessor t of s which is a read operation in form of `x = o.f`.

Similarly, if we consider the second assertion of $(*)$, we can prove that there is a transitive successor t of s which is a read operation in form of `x = o.f`.

Because the r_set' satisfies the properties described in Proposition 1, there must be `o.readLock` $\in r\_set'[s]$.

If $(*)$ does not hold, i.e. $l \in r\_set[s] \setminus r\_set0[s]$, then we know that l is added into $r\_set[s]$ in ExpandLockSets. Since l is a read lock and the Condition 2 of ExpandLockSets does nothing following the assumptions before, we know that l is added into $r\_set[s]$ according to the Condition 1. Thus we know that there is a predecessor t of s in the CFG such that $u.readLock \in r\_set[t]$, $u.readLock \in r\_set[s]$ and $\alpha_P(o, u.f)$. By $u.readLock \in r\_set[t]$ we know that the read or write lock of u must be acquired before s. And by $u.readLock \in r\_set[s]$ we know that the lock of u must be also released before s. And by $\alpha_P(o, u.f)$ we know that o is an alias of a field of u.

Because r_set' satisfies the properties in Proposition 2, there also must be $l =$ `o.readLock` $\in r\_set'[s]$.

In summary, we have $r\_set[s] \subseteq r\_set'[s]$. This concludes the proof. □

Definition 2. *A class (data structure) C is* serializable*, if the result of any concurrent invocations of the methods of C is equivalent to one where invocations appears sequentially*[1]*.*

Serializable is a description of atomicity. Generally speaking, serializability needs the data structure to be monopolized by one thread in execution, which restricts concurrency. For some applications, such as the implementation of a dictionary, it may be too strong to have serializability. Our algorithm does not lock the whole data structure, thus does not ensure serializability, but still gives reasonable result. Moreover, for tree-like data structures (including lists), our algorithm guarantees the modified data structure is serializable.

Definition 3. *Let a_1, a_2 are two concurrent executions of method m, m' of a data structure D, where m and m' may be the same. Let O_1, O_2 are the shared object sets visited in a_1 and a_2, respectively. We call that a_1* precedes *a_2, denoted as $a_1 \prec a_2$, if for every shared object $o \in O_1 \cap O_2$, for every pair of conflict visits on o from a_1 and a_2, respectively, the visit from a_1 happens before*

[1] The definition we use is similar to the conflict-serializable used in [7]. Intuitively, we forbid a read or write thread passes over another write thread in the access of the data structure.

the visit from a_2. Here, we say a pair of visits on object o conflict if at least one of them is a write.

We call a data structure D is in order*, if for any pair of concurrent executions a_1, a_2 of method(s) in D, there are $a_1 \prec a_2$ or $a_2 \prec a_1$.*

Proposition 4. *If the data structure is maintained as a tree, then the generated class by algorithm* MethodTransform *is in order.*

Proof. We prove by induction on the tree.

As the basis, if the data structure is an empty tree, then the concurrent executions of methods do not share data. The data structure is obvious in order.

Suppose that we have proved that the data structure is in order if height of the tree is less than h. Now we prove that it is in order for any tree of height h.

Let a and a' are two concurrent executions of some methods m and m' in the different threads, where m and m' may be the same. Without losing generality, we suppose that a visit the root object r of the tree first. By Proposition 1, a holds the read or write lock of r while it visits r.

On one hand, if a holds the write lock of r, then a' cannot visit r until a completes the visit of r, i.e. the visit of r in a precedes any visit of r in a'. By Proposition 2, before a completes the visit of r and release the write lock, either a holds a root of subtree r' of r and acquires its lock, or a completes its work and exits. Because the height of r' is less than r, we know that a precedes a' for the subtree, by the induction hypothesis.

On the other hand, if a holds only the read lock of r, then a will not write any node in the subtree of r by Proposition 2. And a' can either acquire the read lock, or acquire the write lock of r. If a' acquires the read lock of r, it will not write any node in the subtree of r by Proposition 2. Then both a and a' are read-only executions for the tree with root r, then $a \prec a'$ and $a' \prec a$. If a' acquires the write lock of r, a' would be blocked until a completes the visit of r and releases the read lock of r. Then either a holds a root of subtree r' of r and acquires its lock, or a completes its work and exits, by induction hypothesis we conclude a precedes a' for the subtree.

Taking these together, we can conclude that a precedes a' for the whole tree. This completes the proof. □

Theorem 1. *If the data structure is maintained as a tree, then generated class by algorithm* MethodTransform *is serializable.*

Proof. This can be proved by induction on the number of concurrent invocations.

As the basis, the data structure is obviously serializable if there is only one invocation.

We suppose that for any $n-1$ concurrent invocations, the data structure is serializable. Now we consider n invocations a_1, ..., a_n. By Proposition 4, the data structure is in order. Then we can suppose that $a_1 \prec \cdots \prec a_n$. Because $a_1 \prec a_2, \ldots, a_n$, the states of each object in the data structure visited by a_1 is the same as the result if we perform a_1 without $a_2, \ldots, a_n$. Then we know the result of the n transactions is the same as the result if we perform a_1 separately first, then perform the remaining transactions $a_2, \ldots, a_n$ concurrently. Then by induction hypothesis, the result is the same as we perform $a_1, a_2, \ldots, a_n$ sequentially. □

Table 3: The Experimental Results

Name	Line of Code	Generated Locks	Time (s)
AvlTree	347	28	1.0
BarnesHut	394	20	0.7
LinedList	983	38	1.2
TreeMap	2443	81	2.1

And for a tree-like data structure, visits to the tree nodes are always from the root to the leaves. This partial order of visits implies the order of locking. Therefore, we have

Theorem 2. *If the data structure is maintained as a forest, then generated class by algorithm* MethodTransform *is deadlock free.* □

6 Implementation and Experiments

We have implemented our algorithms as a prototype tool based on Soot [23], which is a widely-used framework for analyzing and transforming Java and Android applications. We feed the Java source code into our tool. It calls Soot to translate the source to Shimple, which is the Soot internal language in the form of SSA-style typed 3-address representation. Our tool then works on the analyses and transformations. The analyses and automatic locking insertion are performed on Shimple. Finally, the tool uses Soot to output directly the Java bytecode with fine-grain locking statements inserted, without touching the original source code. The whole process is automatic. If necessary, we can also decompile the result bytecode to Java code for debugging.

The points-to analysis facility provided by the Soot tool is used in our implementation. It shows that a simple points-to analysis is accurate enough in our intra-procedure algorithms, which will not be a performance bottleneck.

6.1 Experiments on Some Samples

Besides the artificial examples as showed in section 2, we run our tool on some real world tree-like data structures, as listed in Table 3. Here the AvlTree is an AVL tree data structure taken from a textbook; BarnesHut is the core component of an n-body problem simulator; LinkedList and TreeMap are two container classes taken from JDK with minor modifications.

We ran all the experiments on a machine with 4 hardware threads. Specially, we used an Intel Core i5 processor with 4 cores at 3.0GHz. Table 3 lists the performance of our tool, where the 2nd column gives the LOC of the original programs, the 3rd column shows the number of lock instructions inserted, and the last column lists time consumed by our tool to preform the analysis and transformation. All the work for these sample programs can be done within a few seconds, that shows our approach is practically usable.

6.2 Comparison of Different Approaches

One motivation of this work is that the widely used `synchronized` discipline in OO might result low parallelism in many circumstances, where fine-grain locking scheme need to considered, especially for the concurrent data structures. Now we develop an approach, which makes the fine-grain locking automatic and affordable. However, the problem remains: can the fine-grain locking scheme really bring some performance benefits? After finished the work, we need to give some results to support our opinion. To make the comparison more informative, we include also the result of the manual hand-over-hand locking approach and the Domination Locking proposed in [7] in our experiment.

6.2.1 Methodology

We make the experiment for one typical data structure, linked lists. We consider only three common operations: *push*, *pop* and *lookup*. Pushes and pops are performed at the head of the list, which are the special case of insertion and deletion without lookup.

In the evaluation, we assume that the data structure is used under the workload of 20% pushes, 10% pops, and 70% lookups. At the beginning, we put 10000 integers in the list, which are generated from a random uniform distribution between 1 and 20000.

To ensure consistent and trustworthy of the results, each experiment consists of five passes: the first pass warms up the JVM and the following four passes are timed. In each pass, each thread in the test program performs 10000 randomly chosen operations on the list. In addition, a new data structure is used for each pass. The arithmetic average of the throughput of the times of the passes is recorded as the final result.

6.2.2 Evaluation Results

Figure 11 shows the results of our experiments. Here we compare the performances of our automatic fine-grain locking to coarse-grain `synchronized` methods, Domination Locking [7], and manual hand-over-hand exclusive locking. Especially, the figure shows the throughout of total list operations per second for one to four threads. The vertical axis gives the throughout, where the horizontal axis is for the numbers of threads.

From the figure, we can see clear evidence to support our opinion. For the single thread case, our fine-grain locking version runs some slower than the coarse-grain `synchronized` method version and manual hand-over-hand locking version, because it performs many readers-writer lock operations during the execution. The Domination Locking is the slowest because of the runtime overhead. Following the increasing of the threads, even for two threads, our fine-grain version becomes superior than other versions. In addition, the throughout for our fine-grain version increases stably following in the number of thread[2].

For the `synchronized` implementation, we see that the throughout gets lower and lower when the threads get more. This may be caused by that all methods should run sequentially when executes the list operations.

[2]Of course, it is limited by the cores of the computer.

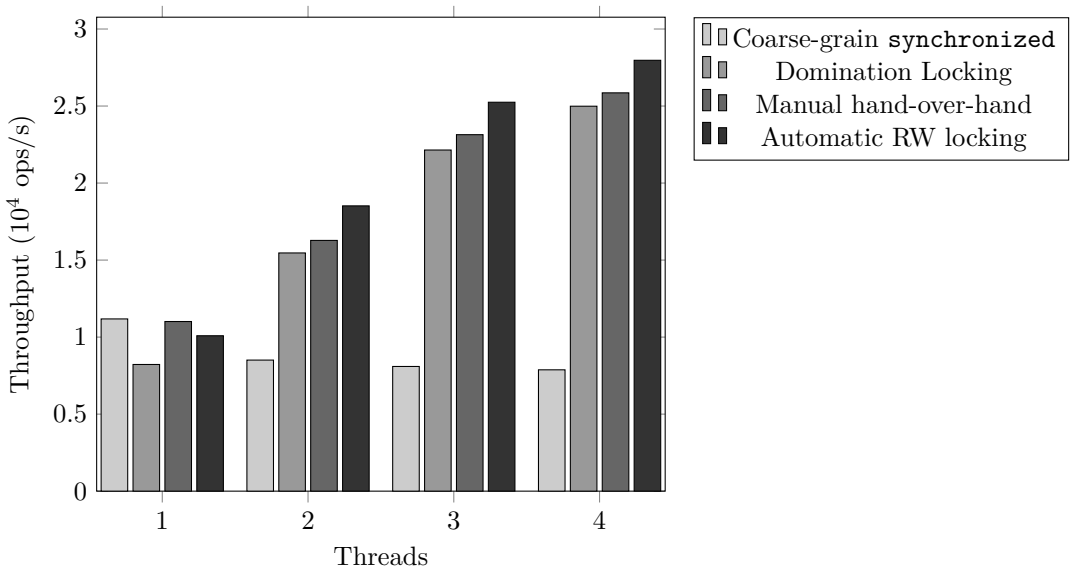

Figure 11: Throughput for a concurrent linked list: comparison among a `synchronized` implementation, Domination Locking [7], manual hand-over-hand exclusive locking, and our automatic fine-grain locking using readers-writer locks.

Our approach uses readers-writer lock, while the Domination Locking version and manual hand-over-hand locking version use exclusive locks. This makes our approach outperforms slightly the other two versions.

As a conclusion, since the fine-grain readers-writer locks give more possibility for the parallel execution, our automatic-produced locking version scales very well, and outperforms the `synchronized` approach as the number of thread increases. This result is very encouraging.

The fine-grain locking gains performance benefits in a variety of aspects:

1. Each object has its own lock, which enables more concurrent execution than coarse-grain locking on the entire data structure.

2. Our algorithm gives the minimal locking range in the code, which increases parallelism further. This benefit is more remarkable for those time-consuming operations, for which, the `synchronized` strategy may lock the entire data structure for a very long period.

3. Compared with the regular locks, readers-writer locks used in our work allow more concurrent read operations.

On the other hand, our fine-grain locking brings more overhead for slightly heavier readers-writer locks and more frequent locking operations. Luckily, the cost of fine-grain locking does not exceed the benefits most of the times. Our preliminary experiments give us some substantial confidences.

If the manual hand-over-hand locking uses readers-writer locks, it may gain the similar performance compared to our approach. However, the manual fine-grain locking is tedious and error-prone. Our automatic approach is rather handy.

Domination Locking is a both automatic and fine-grain approach. However, compared to our approach, Domination Locking does not utilize the benefit of readers-writer locks, and the runtime overhead also slows down the result.

7 Related Work

Now we overview some related work and make some comparisons.

7.1 Domination Locking in Forest-Based Modules

The main aim of our work is similar to Golan-Gueta et al. [7]. These authors proposed the *Domination Locking* (DL) protocol, that is a dynamic technique for automatically adding fine-grain locks to abstract data types when the heap graph is guaranteed to be a forest. In addition, they also tried to handle temporary violation of the forest shape constraints.

In DL, heap objects are divided in two categories: exposed objects which may be returned to the outside of the module, and hidden objects. DL requires the exposed objects to be locked at the very beginning of the operations to ensure atomicity, and a strict locking order is forced for the exposed objects to prevent deadlocks. On the other hand, hidden objects are protected by the related exposed objects.

Although DL is a general locking protocol, Golan-Gueta et al. require that the data structures to be handled conform a rather restricted forestness condition for applying the protocol. And furthermore, this technique needs run-time information bookkeeping for the exposedness and two kinds of reference counters, which will bring visible overheads.

To maintain forestness, it is impossible to return a sub-tree and make the root of the sub-tree exposed in this dynamic technique, without cutting it off from the data structures. This restricts the flexibility of the modules. As a result, in practice, the programs that can be handled by dynamic techniques are not distinctly more than the ones that can be handled by static techniques.

In contrast, our algorithms may be more effective in several aspects:

1. Our techniques are fully static without runtime bookkeeping.
2. Our algorithms may produce more fine-grain result since the generated locking range is minimum for many cases.
3. Our techniques utilize readers-writer locks to increase parallelism.

7.2 Other Related Work

While our work proposes a general locking insertion approach, the related work is usually about implementing atomic sections. Transactional memory [10] is popular for implementing atomic sections. However, recent work takes more attention to statically inferring locks for atomic and deadlock-free execution.

There has been some work on inferring locks for implementing atomic sections. Most of the algorithms infer locks for following the two-phase locking (2PL) protocol [6], in which locking is done in two phases, in the first phase locks are only allowed to be acquired, in the second phase locks are only allowed

to be released. These restrictions prevent early releasing a lock of an object when it will not be used, thus it greatly limits parallelism for long executions, such as tree traversal. In contrast, our approach does not have these restrictions and computes the minimal safe lock set, which may enable more parallelism.

The Autolocker tool [19] developed by MacCloskey et al. is not totally automatic. It needs programmers to annotate which locks protect each path expression. Emmi et al. [5] extend the power of Autolocker by removing the manual annotations. They proposed an algorithm which works in the presence of pointers and procedures, and sets up the lock allocation problem as a 0-1 integer linear programming which minimizes the conflict cost between atomic sections while simultaneously minimizing the number of locks. These algorithms employ a 2PL variant in which all locks are released at the end of transaction, which may limits parallelism as described before.

The work of Hicks et al. [11] proposed an approach to infer abstract objects that are each protected by their own lock. However, per-instance locks are not supported in this work, and locks are acquired at the start and released at the end of atomic sections. These will exclude much parallelism, as we discussed before. On the other hand, Cherem et al. [3] suggested a lock inferences method for object instances. However, in the method, locks are also acquired at the start of the atomic sections and released at the end. In another work, Gudka et al. [9] showed how the external libraries can be handled in automatic insertion. Their method also uses a customized 2PL protocol, which implies less parallelism.

Liu and Zhang [18] use Multiple Granularity Lock (MGL) to improve parallelism. They calculate the abstract object graph from the data structure, tag the graph edges (object fields), and produce the lock operations. Their framework is applicable to general-purpose programs, and the MGL is expected to outperform exclusive locks and readers-writer locks. Kalikar and Nasre [15] proposed DomLock technique for hierarchical data structures. DomLock uses MGL, and defines the concept of dominator which is similar to Domination Locking. DomLock is also applicable to DAGs and cycles. However, since the shape of data structures is more flexible, MGL and DomLock still requires variants of 2PL protocol, which may perform less efficiently for tree shape structures.

There are also non-2PL techniques for automatic lock insertion. Based on the static concurrent analysis by Kahlon et al. [14], Kahlon [13] proposed an automatic technique to add exclusive locks to concurrent programs to fix bugs like atomicity violations. The technique guarantees atomicity and deadlock freedom, and keeps the newly introduced critical sections as small as possible. Our approach has similar motivation, which computes the minimal locking range and use locking range expansion to prevent deadlocks for tree data structures. However, unlike Kahlon's approach, our approach uses readers-writer locks, which increases parallelism and also brings much complexity. Moreover, our approach utilizes the shape properties to get more precise result, and the approach can deal with object-oriented programs with complex references.

Boyapati et al. [2] showed that a specific ownership type system can be used to prevent data race and deadlocks, but it cannot guarantee serializability (atomicity). Moreover, mutable shared data and synchronization are not allowed, locks are not used at all, which much restricts the use of the type system. Our work is inspired by the intuition of dynamic ownership by Leino et al. [17, 4], which allows more flexible encapsulation rules than ownership types. In dynamic ownership, objects can be explicitly unpacked for mutations, then

can be packed again for encapsulation. This is similar to acquire and release the lock of the object. Our program analysis techniques are similar to the algorithm by Hu et al. [12] used for analyzing dynamic ownership.

8 Conclusion

The correct application of fine-grain locking is challenging and error-prone. Consequently, programmers often resort to coarse-grain locking, that often leads to lower performance and less scalability.

Based on the program analysis technique, this paper presents a static approach which can automatically insert fine-grain locks into tree-shape shared data structures in object-oriented programs. In the approach, a two-stage dataflow analysis is used to trace the access of each object and figure out the locking states. Then, it tries to determine all the positions where locks should be acquired or released and finally inserts the corresponding lock-related statements automatically. The result data structures produced by our approach are guaranteed data race free and minimum in the locked ranges. In addition, we prove that for the tree-shape structures, the transformation ensures serializability and deadlock freedom. Some experiments illustrate that our approach is practically useful. It can help programmers to gain the benefits of the fine-grain locking, and reduces their workload greatly.

As future work, we consider extending this work to deal with more general hierarchical data structures.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition edition, September 2006. 00049.

[2] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM. 00000.

[3] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring Locks for Atomic Sections. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 304–315, New York, NY, USA, 2008. ACM.

[4] Werner Dietl and Peter Müller. Object ownership in program verification. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 289–318. Springer, 2013.

[5] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock Allocation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 291–296, New York, NY, USA, 2007. ACM.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.

[7] Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic Fine-grain Locking Using Shape Properties. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 225–242, New York, NY, USA, 2011. ACM.

[8] James Gosling, Bill Joy, Guy L. Steele Jr, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, Upper Saddle River, NJ, 1 edition edition, May 2014.

[9] Khilan Gudka, Tim Harris, and Susan Eisenbach. Lock Inference in the Presence of Large Libraries. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, number 7313 in Lecture Notes in Computer Science, pages 308–332. Springer Berlin Heidelberg, June 2012. DOI: 10.1007/978-3-642-31057-7_15.

[10] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[11] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. June 2006.

[12] Tingting Hu, Haiyang Liu, Ke Zhang, and Zongyan Qiu. *Automatic Annotating and Checking of Dynamic Ownership*, pages 78–94. Springer International Publishing, Cham, 2016.

[13] V. Kahlon. Automatic lock insertion in concurrent programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2012*, pages 16–23, 2012.

[14] Vineet Kahlon, Franjo Ivančić, and Aarti Gupta. Reasoning About Threads Communicating via Locks. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, number 3576 in Lecture Notes in Computer Science, pages 505–518. Springer Berlin Heidelberg, July 2005. DOI: 10.1007/11513988_49.

[15] Saurabh Kalikar and Rupesh Nasre. DomLock: A New Multi-granularity Locking Technique for Hierarchies. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 23:1–23:12, New York, NY, USA, 2016. ACM.

[16] William Alexander Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, New Brunswick, NJ, USA, 1992. UMI Order No. GAX92-19944.

[17] K. Rustan M. Leino and Peter Müller. Object Invariants in Dynamic Contexts. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 491–515. Springer Berlin Heidelberg, January 2004.

[18] Peng Liu and Charles Zhang. Unleashing Concurrency for Irregular Data Structures. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 480–490, New York, NY, USA, 2014. ACM.

[19] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 346–358, New York, NY, USA, 2006. ACM. 00145.

[20] John Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

[21] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[22] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, Boston, 4th edition, March 2014.

[23] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13, Mississauga, Ontario, Canada, 1999. IBM Press. 00765.