Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets

Tayfun Elmas
Koç University

Shaz Qadeer
Microsoft Research

Serdar Tasiran
Koç University

November 17, 2006

Technical Report
MSR-TR-2006-163

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Goldilocks: Efficiently Computing the Happens-Before Relation Using Locksets

Tayfun Elmas
Koç University

Shaz Qadeer
Microsoft Research

Serdar Tasiran
Koç University

Abstract

We present a new lockset-based algorithm, Goldilocks, for precisely computing the happens-before relation and thereby detecting data-races at runtime. Dynamic race detection algorithms in the literature are based on vector clocks or locksets. Vector-clock-based algorithms precisely compute the happens-before relation but have significantly more overhead. Previous lockset-based race detection algorithms, on the other hand, are imprecise. They check adherence to a particular synchronization discipline, i.e., a sufficient condition for race freedom and may generate false race warnings. Our algorithm, like vector clocks, is precise, yet it is efficient since it is purely lockset based.
We have implemented our algorithm inside the Kaffe Java Virtual Machine. Our implementation incorporates lazy evaluation of locksets and certain "short-circuit checks" which contribute significantly to its efficiency. Experimental results indicate that our algorithm's overhead is much less than that of the vector-clock algorithm and is very close to our implementation of the Eraser lockset algorithm.

1 Introduction

Race conditions on shared data are often symptomatic of a bug and their detection is a central issue in the functional verification of concurrent software. Numerous techniques and tools have been developed to analyze races and to guard against them [14, 18, 6, 1]. These techniques can be broadly classified as static and dynamic. Some state-of-the-art tools combine techniques from both categories. This paper is about a dynamic race detection algorithm.

Algorithms for runtime race detection make use of two key techniques: locksets and vector clocks. Roughly speaking, lockset-based algorithms compute at each point during an execution for each shared variable q a set $LS(q)$. The lockset $LS(q)$ consists of the locks and other synchronization primitives that, according to the algorithm, protect accesses to q at that point. Typically, $LS(q)$ is a small set and can be updated relatively efficiently during an execution. The key weakness of lockset-based algorithms in the literature is that they are specific to a particular locking discipline which they try to capture directly in $LS(q)$. For instance, the classic lockset algorithm popularized by the Eraser tool [14], is based on the assumption that each potentially shared variable must be protected by a single lock throughout the whole computation. Other similar algorithms can handle more sophisticated locking mechanisms [1] by incorporating knowledge of these mechanisms into the lockset inference rules. Still, lockset-based algorithms based on a particular synchronization discipline have the fundamental shortcoming that they may report false races when this discipline is not obeyed. Vector-clock [10] based race detection algorithms, on the other hand, are precise, i.e., declare a race exactly when an execution contains two accesses to a shared variable that are not ordered by the happens-before relation. However, they are significantly more expensive computationally than lockset-based algorithms as argued and demonstrated experimentally in this work.

In this paper we provide, for the first time, a lockset-based algorithm, Goldilocks, that precisely captures the happens-before relation. In other words, we provide a set of lockset update rules and formulate a *necessary and sufficient* condition for race-freedom based solely on locksets computed using these rules. Goldilocks combines the precision of vector clocks with the computational efficiency of locksets. We can uniformly handle a variety of synchronization idioms such as thread-local data that later becomes shared, shared data protected by different locks at different points in time, and data protected indirectly by locks on container objects.

For dynamic race detection tools used for stress-testing concurrent programs, precision may not be desired or necessary. One might prefer an algorithm to signal a warning about not only about races in the execution being checked, but also about "feasible" races in similar executions [11]. It is possible to incorporate this kind of capability into our algorithm by slightly modifying the lockset update rules or the race condition check. However, the target applications for our race detection algorithm are continuous monitoring for actual races during early development and deployment, and partial-order reduction during model checking as is done in [7]. False alarms and reports of feasible rather than actual races unnecessarily interrupt execution and take up developers' time in the first application and cause computational inefficiency in the latter. For these reasons, for the targeted applications, the precision of our algorithm is a strength and not a weakness.

We present an implementation of our algorithm that incorporates lazy computation of locksets and "short circuit checks": constant time sufficient checks for race freedom. These implementation improvements contribute significantly to the computational efficiency of our technique and they appear not to be applicable to vector clocks. We implemented our race-detection algorithm in C, integrated with the Kaffe Java Virtual Machine [17]. An important

contribution of this paper is an experimental comparison of the Goldilocks algorithm with the vector-clock algorithm and our implementation of the Eraser algorithm. We demonstrate that our algorithm is much more efficient than vector clocks and about as efficient as Eraser.

This paper is organized as follows. Section 2 describes the Goldilocks algorithm and presents an example which contrasts our algorithm with existing locksets algorithms. Section 3 explains the implementation of our algorithm in the Kaffe JVM. Experimental evaluation of our algorithm is presented in Section 4. Related work is discussed in Section 5.

2 The Goldilocks algorithm

In this section, we describe our algorithm for checking whether a given execution σ has a data-race. We use the standard characterization of data-races based on the happens-before relation, i.e., there is a data race between two accesses to a shared variable if they are not ordered by the happens-before relation. The happens-before relation for an execution is defined by the memory model. We use a memory model similar to the Java memory model [9] in this paper. Our algorithm is sound and precise, that is, it reports a data-race on an execution iff there is a data-race in that execution.

2.1 Preliminaries

A state of a concurrent program consists of a set of local variables for each thread and a set of global objects shared among all threads. Let *Tid* be the set of thread identifiers and *Addr* be the set of object identifiers. Each object has a finite collection of fields. *Field* represents the set of all fields. and is a union of two disjoint sets, the set *Data* of *data* fields and the set *Volatile* of *volatile* fields. A *data variable* is a pair (o, d) of an object o and a data field d. A *synchronization variable* is a pair (o, v) of an object o and a volatile field v. A concurrent execution σ is represented by a finite sequence $s_1 \xrightarrow{\alpha_1}_{t_1} s_2 \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} s_{n+1}$, where s_i is a program state for all $i \in [1 \ldots n+1]$ and α_i is one of the following actions for all $i \in [1 \ldots n]$: $acq(o)$, $rel(o)$, $read(o, d)$, $write(o, d)$, $read(o, v)$, $write(o, v)$, $fork(u)$, $join(u)$, and $alloc(o)$. We use a linearly-ordered sequence of actions and states to represent an execution for ease of expressing the lockset-update rules and the correctness of the algorithm. This sequence can be *any* linearization of the union of the following partial orders defined in [9]: (i) the program order for each thread and (ii) the synchronizes-with order for each synchronization variable. The particular choice of the linearization is immaterial for our algorithm. In our implementation (Section 3) each thread separately checks races on a (linearly-ordered) execution that represents its view of the evolution of program state.

The actions $acq(o)$ and $rel(o)$ respectively acquire and release a lock on object o. There is a special field $l \in Volatile$ containing values from $Tid \cup \{null\}$ to model the semantics of an object lock. The action $acq(o)$ being performed by thread t blocks until $o.l = null$ and then atomically sets $o.l$ to t. The action $rel(o)$ being performed by thread t fails if $o.l \neq t$, otherwise it atomically sets $o.l$ to $null$. Although we assume non-reentrant locks for ease of exposition in this paper, our algorithm is easily extended to reentrant locks. The actions $read(o, d)$ and $write(o, d)$ respectively read and write the data field d of an object o. A thread *accesses* a variable (o, d) if it executes either $read(o, d)$ or $write(o, d)$. Similarly, the actions $read(o, v)$ and $write(o, v)$ respectively read and write the volatile field v of an object o. The action $fork(u)$ creates a new thread with identifier u. The action $join(u)$ blocks until the thread with identifier u terminates. The action $alloc(o)$ allocates a new object o. Of course, other actions (such as arithmetic computation, function calls, etc.) also occur in a real execution but these actions are irrelevant for our exposition and have consequently been elided.

Following the Java Memory Model [9], we define the happens-before relation for a given execution as follows.

Definition 1 *Let $\sigma = s_1 \xrightarrow{\alpha_1}_{t_1} s_2 \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} s_{n+1}$ be an execution of the program. The happens-before relation $\xrightarrow{hb}$ for σ is the smallest transitively-closed relation on the set $\{1, 2, \ldots, n\}$ such that for any k and l, we have $k \xrightarrow{hb} l$ if $1 \leq k \leq l \leq n$ and one of the following holds:*

1. $t_k = t_l$.
2. $\alpha_k = rel(o)$ *and* $\alpha_l = acq(o)$.
3. $\alpha_k = write(o, v)$ *and* $\alpha_l = read(o, v)$.
4. $\alpha_k = fork(t_l)$.
5. $\alpha_l = join(t_k)$.

We use the happens-before relation to define data-race free executions as follows. Consider a data variable (o, d) in the execution σ. The execution σ is *race-free* on (o, d) if for all $k, l \in [1, n]$ such that $\alpha_k, \alpha_l \in \{read(o, d), write(o, d)\}$, we have $k \xrightarrow{hb} l$ or $l \xrightarrow{hb} k$. For now, our definition does not distinguish between read and write accesses. We are currently refining our algorithm to make this distinction in order to support concurrent-read/exclusive-write schemes.

2.2 The algorithm

Our algorithm for detecting data races in an execution σ uses an auxiliary map LS from $(Addr \times Data)$ to $Powerset((Addr \times Volatile) \cup Tid)$. This map provides for each data variable (o, d) its lockset $LS(o, d)$ which contains volatile variables, some of which represent locks and thread identifiers. The algorithm updates LS with the execution of each transition in σ. The set of rules for these updates are shown in Figure 1. Initially, the partial map LS is empty. When an action α happens, the map LS is updated according to the rules in the figure.

Goldilocks maintains for each lockset $LS(o, d)$ the following invariants: 1) If $(o', l) \in LS(o, d)$ then the last access to (o, d) happens-before a subsequent $acq(o')$. 2) If $(o', v) \in LS(o, d)$ then the last access to (o, d) happens-before a subsequent $read(o', v)$. 3) If $t \in LS(o, d)$ then the last access to (o, d) happens-before any subsequent action by thread t. The first two invariants indicate that $LS(o, d)$ contains the locks and volatile variables whose acquisitions and reads, respectively, create a happens-before edge from the last access of (o, d) to any subsequent access of (o, d), thereby preventing a race. As a result of the last invariant, if $t \in LS(o, d)$ at an access to a data variable (o, d) by thread t, then the previous access to (o, d) is related to this access by the happens-before relation. A race on (o, d) is reported in Rule 1, if $LS(o, d) \neq \emptyset$ and $t \notin LS(o, d)$ just before the update.

1. $\alpha \in \{read(o,d), write(o,d)\}$:

   if $LS(o,d) \neq \emptyset$ and $t \notin LS(o,d)$
   report data race on (o,d)
   $LS(o,d) := \{t\}$

2. $\alpha = read(o,v)$:

   foreach (o',d):
   if $(o,v) \in LS(o',d)$ add t to $LS(o',d)$

3. $\alpha = write(o,v)$:

   foreach (o',d):
   if $t \in LS(o',d)$ add (o,v) to $LS(o',d)$

4. $\alpha = acq(o)$:

   foreach (o',d):
   if $(o,l) \in LS(o',d)$ add t to $LS(o',d)$

5. $\alpha = rel(o)$:

   foreach (o',d):
   if $t \in LS(o',d)$ add (o,l) to $LS(o',d)$

6. $\alpha = fork(u)$:

   foreach (o',d):
   if $t \in LS(o',d)$ add u to $LS(o',d)$

7. $\alpha = join(u)$:

   foreach (o',d):
   if $u \in LS(o',d)$ add t to $LS(o',d)$

8. $\alpha = alloc(x)$:

   foreach $d \in Data$: $LS(x,d) := \emptyset$

Figure 1: The lockset update rules for the Goldilocks algorithm (action α executed by thread t)

We now present the intuition behind our algorithm. Let (o,d) be a data variable, α be the last access to it by a thread a, and β be the current access to it by thread b. Then α happens-before β if there is a sequence of happens-before edges connecting α to β. The rules in Figure 1 are designed to compute the transitive closure of such edges. When α is executed, the lockset $LS(o,d)$ is set to the singleton set $\{a\}$. This lockset grows as synchronizing actions happen after the access. The algorithm maintains the invariant that a thread identifier t is in $LS(o,d)$ iff there is a sequence of happens-before edges between α and the next action performed by thread t. The algorithm adds a thread identifier to $LS(o,d)$ as soon as such a sequence of happens-before edges is established.

Note that each of the rules 2–7 requires updating the lockset of each data variable. A naive implementation of this algorithm would be too expensive for programs that manipulate large heaps. In Section 3, we present a scheme to implement our algorithm by applying these updates lazily.

The following theorem expresses the fact that our algorithm is both sound and precise.

Theorem 1 (Correctness) *Consider an execution* $\sigma = s_1 \xrightarrow{\alpha_1}_{t_1} s_2 \cdots s_n \xrightarrow{\alpha_n}_{t_n} s_{n+1}$ *and let* LS_i *be the value of the lockset map* LS *as computed by the Goldilocks algorithm when* σ *reaches state* s_i. *Let* (o,d) *be a data variable and* $i \in [1, n-1]$ *be such that* α_i *and* α_n *access* (o,d) *but* α_j *does not access* (o,d) *for all* $j \in [i+1, n-1]$. *Then* $t_n \in LS_n(o,d)$ *iff* $i \xrightarrow{hb} n$.

The proof appears in the appendix.

Our algorithm has the ability to track happens-before edges from a write to a subsequent read of a volatile variable. Therefore, our algorithm can handle any synchronization primitive, such as semaphores and barriers in the `java.util.concurrent` package of the Java standard library, whose underlying implementation can be described using a collection of volatile variables.

Goldilocks can also handle the happens-before edges induced by the wait/notify mechanism of Java without needing to add new rules. The following restrictions of Java ensure that, for an execution the happens-before relation computed by our lockset algorithm projected onto data variable accesses remains unchanged even if the wait/notify synchronization adds new happens-before edges: 1) Each call to `o.wait()` and `o.notify()` be performed while holding the lock on object `o`. 2) The lock of `o` released when `o.wait()` is entered and it is again acquired before returning from `o.wait()`.

2.3 Example

In this section, we present an example of a concurrent program execution in which lockset algorithms from the literature declare a false race while our algorithm does not. The lockset algorithms that we compare ours with are based on the Eraser algorithm [14], which is sound but not precise.

The pseudocode for the example is given below. The code executed by each thread `Ti` is listed next to `Ti:`.

```
Class IntBox {  Int x; }

IntBox a = new IntBox();  // IntBox object o1 created
IntBox b = new IntBox();  // IntBox object o2 created

  T1:  acq(L1); a.x++; rel(L1);
  T2:  acq(L1); acq(L2);
       tmp = a; a = b; b = tmp;
       rel(L1); rel(L2);
  T3:  acq(L2); b.x++; rel(L2);
```

In this example, two `IntBox` objects `o1` and `o2` are created and locks `L1` and `L2` are used for synchronization. The program follows the convention that `L1` protects accesses to `a` and `a.x`, similarly, `L2` protects accesses to `b` and `b.x`. At all times, each `IntBox` object and its integer field `x` are protected by the same lock. `T2` swaps the objects referred to by the variables `a` and `b`.

Consider the interleaving in which all actions of `T1` are completed, followed by those of `T2` and then `T3`. `T2` swaps the objects referred to by variables `a` and `b` so that during `T3`'s actions `b` refers to `o1`. `o1.x` is initially protected by `L1` but is protected by `L2` after `T2`'s actions are completed.

The most straightforward lockset algorithm is based on the assumption that each shared variable is protected by a fixed set of locks throughout the execution. Let $LH(t)$

represent the set of locks held by thread t at a given point in an execution. This algorithm attempts to infer this set by updating $LS(o,d)$ to be the intersection $LH(t) \cap LS(o,d)$ at each access to (o,d) by a thread t. If this intersection becomes empty, a race is reported. This approach is too conservative since it reports a false race if the lock protecting a variable changes over time. In the example above, when `T3` accesses `b.x`, the standard lockset algorithm declares a race since $LS($`o1.x`$) = \{$`L1`$\}$ (`b` points to `o1`) before this access and `T3` does not hold `L1`.

A less conservative alternative is to update $LS(o,d)$ to $LH(t)$ rather than $LH(t) \cap LS(o,d)$ after a race-free access to (o,d) by a thread t. For any given execution, this strategy, just like the previous strategy, will report a data-race if there is one but is still imprecise and might report false races. In the example above, this approach is unable to infer the correct new lockset for `o1.x` after `T2`'s actions are completed. This is because `T2` does not directly access `o1.x` and, as a result, $LS($`o1.x`$)$ is not modified by `T2`'s actions.

Variants of lockset algorithms in the literature use additional mechanisms such as a state machine per shared variable in order to handle special cases such as thread locality, object initialization and escape. However these variants are neither sound nor precise, and they all report false alarms in scenarios similar to the one in the example above.

Our algorithm's lockset update rules allow a variable's locksets to grow and change during the execution. The lockset of a variable may be modified even without the variable being accessed. In this way, we are able handle dynamically changing locksets and ownership transfers and avoid false alarms. In the example above, the lockset of `o1.x` evolves with our update rules during the execution as illustrated in Figure 2.

The vector-clock algorithm does not declare a false race in this example and similar scenarios. However, as discussed in Section 3, it accomplishes this at significantly increased computational cost compared to our optimized implementation of the lockset update rules.

3 Implementation with lazy evaluation

We implemented the Goldilocks algorithm in Kaffe [17], a clean room implementation of the Java virtual machine in C. Our implementation currently runs in the interpreting mode of Kaffe's runtime engine. The pseudocode is given in Figure 3. There are two important features that contribute to the performance of the algorithm in practice: short-circuit checks and lazy evaluation of lockset update rules. Short-circuit checks are cheap, sufficient checks for a happens-before edge between the last two accesses to a variable. We use short-circuit checks to eliminate unnecessary application of the lockset update rules. Lazy evaluation runs the lockset update rules in Figure 1 only when a data variable is accessed and all the short-circuit checks fail to prove the existence of a happens-before relationship.

There are two reasons we implemented our lockset algorithm lazily: 1) Managing and updating a separate lockset for each data variable have high memory and computational cost. Our lockset rules are expressed in terms of set lookups and insertions, and making the lockset a singleton set with the current thread id after an access. These simple update rules make possible a very easy and efficient form of computing locksets lazily only at an access. 2) For thread-local and well-synchronized variables, there may be no need to run (all of) the lockset update rules, because a short-circuit check or a subset of synchronization actions may be sufficient to show race freedom.

In our way of performing lazy evaluation, we do not explicitly associate a separate lockset $LS(o,d)$ for each data variable (o,d). Instead, $LS(o,d)$ is created temporarily, when (o,d) is accessed and the algorithm, after all short-circuit checks fail, finds it necessary to compute happens-before for that access using locksets. In addition, the lockset update rule for a synchronization action in Figure 1 is not applied to $LS(o,d)$ when the action is performed. We defer the application of these rules until (o,d) is accessed and the lockset update rules are applied for that access. We store the necessary information about a synchronization action in a *cell*, consisting of the current thread and the action. During the execution, cells are kept in a list that we call *update list*, which is represented by its *head* and *tail* pointers in the pseudocode. When a thread performs a synchronization action, it atomically appends its corresponding cell to the update list .

Each variable (o,d) is associated with an instance of *Info*. *info* maps variables to *Info* instances. *info*(o,d) keeps track of three pieces of information necessary to check an access to (o,d): 1) *pos* is a pointer to a cell in the update list (*ref*(*Cell*) is the reference type for *Cell*). 2) *owner* is the identifier of the thread that last accessed (o,d). After each access to (o,d) by thread t, *info*(o,d) is updated so that *pos* is assigned to the reference of the cell at the tail of the update list and *owner* is assigned to t. 3) *alock* is used in a short-circuit check as explained below. Notice that because locksets are created temporarily only when the full checking for the lockset rules is to be done, there is no field of *info*(o,d) that points to a lockset.

We instrumented the JVM code by inserting calls to *Handle-Action*. The procedure *Handle-Action* is invoked each time a thread performs an action relevant to our algorithm. We performed the instrumentation so that the synchronizes-with order and the order of corresponding cells in the update list are kept consistent throughout the execution. Similarly, the order of cells respects the program order of the threads in the execution. We needed only for volatile reads/writes to insert explicit locks to make atomic the volatile access and appending the cell for that action to the update list.

Handle-Action takes as input a thread t and an action α performed by t. If α is a synchronization action, *Handle-Action* appends a cell referring to α to the end of the update list (lines 1-6). If α reads from or writes to a data variable (o,d) and it is the first access to (o,d) it creates a new *Info* for (o,d) and sets its *alock* to one of the locks held by t (lines 8-11). Otherwise, it first runs two short-circuit checks (line 12). If both of the short-circuit checks fail, the procedure *Apply-Lockset-Rules* is called. Before exiting *Handle-Action*, *info*(o,d) is updated to reflect the last access to (o,d) (lines 19-20). *Handle-Action* also garbage collects the cells in the update list that are no longer referenced, by calling *Garbage-Collect-Cells* (line 21).

Apply-Lockset-Rules applies the lockset update rules in Figure 1 but uses a local, temporarily-created lockset $LS(o,d)$. $LS(o,d)$ is initialized to contain *info*(o,d).*owner*, the identifier of the thread that last accessed (o,d), to reflect the effect of Rule 1 for variable accesses. Then the rules for the synchronization actions performed after the last access to (o,d) are applied to $LS(o,d)$ in turn. The cells in the update list between the cell pointed by *info*(o,d).*pos* and the

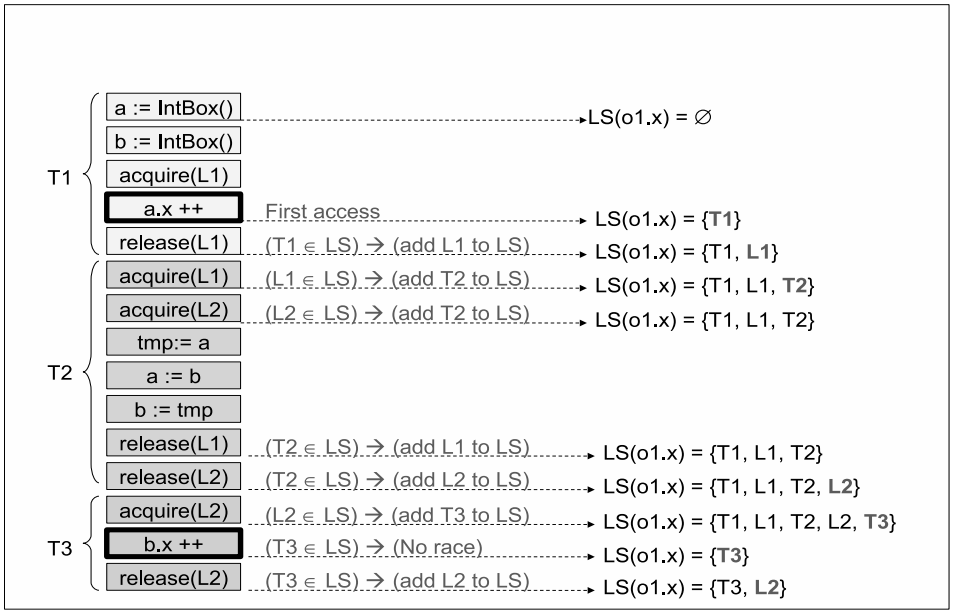

Figure 2: Evaluation of $LS(\texttt{o1.x})$ by Goldilocks.

cell pointed by *tail* are used in this computation. The access causes no warning if the current thread t is added to $LS(o,d)$ by some rule. This check is performed after handling each cell and is also used to terminate the lockset computation before reaching the tail of the update list. If t is not found in $LS(o,d)$, a race condition on (o,d) is reported.

Short-circuit checks: Our current implementation contains two constant time, sufficient checks for the happens--before relation between the last two accesses to a variable (see line 12 of *Handle-Action*). 1) We first check whether the currently accessing thread is the same as the last thread accessed the variable by comparing t and $info(\text{o,d}).owner$. This helps us to handle checking thread local variables in constant time without needing the lockset rules. 2) The second check handles variables that are protected by the same lock for a long time. We keep track of a lock *alock* for each variable (o,d). $info(o,d).alock$ represents an element of $LS(o,d)$ chosen randomly. At the first access to (o,d) $info(o,d).alock$ is assigned one of the locks held by the current thread randomly, or *null* if there is no such lock (line 10). After the next access to (o,d) we check if the lock $info(o,d).alock$ is held by the current thread. If this check fails, $info(o,d).alock$ is reassigned by choosing a new lock (line 15).

Comparison with the vector-clock algorithm: The vector-clock algorithm is as precise as our algorithm. However, the vector-clock algorithm accomplishes this precision at a significantly higher computational cost compared to Goldilocks because lazy evaluation and the short circuit checks make our approach very efficient. This fact is highlighted by the following example. Consider a program with a large number of threads $t_1, ..., t_n$ all accessing the same shared variable (o,d), where all accesses to (o,d) are protected by a single lock l. At each synchronization operation, $acq(l)$ or $rel(l)$, Goldilocks performs a constant-time operation to add the synchronization operation to the update list. Moreover, once $info(o,d).alock = l$, then at each access to (o,d) Goldilocks performs a constant-time look-up to determine the absence of a race. The vector-clock algorithm, on the other hand, maintains a vector of size n for each thread and for each variable. At each synchronization operation, two such vectors are compared element-wise and updated. At each access to (o,d), the vector-clock algorithm performs constant-time work just like Goldilocks. While the vector--clock algorithm does $\Theta(n)$ work for each synchronization operation and $\Theta(1)$ for each data variable access, Goldilocks does $\Theta(1)$ work for every operation. Therefore, Goldilocks is more efficient than the vector-clock algorithm in general. The `SharedSpot` microbenchmark in Section 4 is based on the example described above and the experiments confirm the preceding analysis.

4 Evaluation

In order to evaluate the performance our algorithm, we ran the instrumented version of the Kaffe JVM on a set of benchmarks. In order to concentrate on the races in the applications, we disabled checks for fields of the standard library classes. Arrays were checked by treating each array element as a separate variable. We first present our experiments and discuss their results in Section 4.1.

In order to compare our algorithm with traditional lockset and vector-clock algorithms, we implemented a basic version of the Eraser algorithm that we call Basic-Eraser and a vector-clock based algorithm similar to the one used by Trade [5]. Where possible, we used the same data structure implementations while implementing the three algorithms. For Basic-Eraser, we used the same code for keeping and manipulating locksets that we developed for Goldilocks.

Microbenchmarks: The `Multiset` microbenchmark consists of a number of threads accessing a multiset of integers concurrently by inserting, deleting and querying elements to/from it. The `SharedSpot` benchmark illustrates the case in which a number of integers, each of which is protected by

```
record Cell {                    record Info {
    thread: Tid;                     pos: ref(Cell);
    action: Action;                  owner: Tid;
    next: ref(Cell);}                alock: Addr; }

head, tail: ref(Cell);           info: (Addr × Data) ⟶ Info;

Initially head := new Cell; tail := head; info := EmptyMap;

Handle-Action (t, α):
1  if (α ∈ {acq(o), rel(o), fork(u), join(u), read(o, v), write(o, v),
           finalize(x), terminate(t)}) {
2      tail→thread := t;
3      tail→action := α;
4      tail→next := new Cell;
5      tail := tail→next;
6  }
7  else if (α ∈ {read(o, d), write(o, d)}) {
8      if (info(o, d) is not defined) { //initialize info(o, d) for the first access to (o, d)
9          info(o, d) := new Info;
10         info(o, d).alock := (choose randomly a lock held by t, if any exists);
11     } else {
12         if ((info(o, d).owner ≠ t) ∧ (info(o, d).alock is not held by t)) {
13             Apply-Lockset-Rules (t, (o, d)); // run the lockset algorithm
14             // because short circuits failed, reassign the random lock for (o, d)
15             info(o, d).alock := (choose randomly a lock held by t, if any exists);
16         }
17     }
18     // reset info(o, d) after each access to (o, d)
19     info(o, d).owner := t;
20     info(o, d).pos := tail;
21     Garbage-Collect-Cells (head, tail);
22 }
```

Figure 3: Implementation of the Goldilocks algorithm

a separate unique lock, are accessed concurrently by a number of threads for applying arithmetic operations on them. The `LocalSpot` benchmark is similar to `SharedSpot` but each variable is thread-local. We ran experiments parameterizing the microbenchmarks with the number of threads starting from 1 and doubling until 256. Figure 4 plots for three algorithms the average time spent for checking each variable access against increasing number of threads.

Large benchmarks: We used six benchmark programs commonly used in the literature to compare the performance of the three algorithms on large programs: `Raja`[1] is a ray tracer ($\approx 6K$ *lines*). `SciMark`[2] is a composite Java benchmark consisting of five computational kernels (≈ 2300 *lines*). Four of our benchmarks are from the Java Grande Forum Benchmark Suite, which can be obtained at http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/threads.html. They are `moldyn`, a molecular dynamics simulation (≈ 650 *lines*), `raytracer`, a 3D ray tracer (≈ 1200 *lines*), `montecarlo`, a Monte Carlo simulation ($\approx 3K$ *lines*) and `sor`, a successive over-relaxation program (≈ 220 *lines*).

Table 1 presents the performance statistics of the three algorithms on the benchmark programs. The purpose of this batch of experiments is to contrast the overhead that each of the three approaches incur while checking for races. In this batch of experiments, race checking for a variable was *not* turned off after detecting a race on it, as would be the case in normal usage of a race detection tool. The purpose of this was to enable a fair comparison between algorithms. On this set of benchmarks, Basic-Eraser conservatively declared false races on many variables early in the execution. If race checking on these variables were turned off after Basic-Eraser detects a race on them, Basic-Eraser would have ended up doing a lot less work and checking a lot fewer accesses than the other two approaches, especially since these variables are typically very likely to have races on them later in the execution as well. This would have made the overhead numbers difficult to compare. In Table 1, we give the number of threads created in each program below the name of the benchmark. The column titled "Uninstrumented" reports the total runtime of the program in the uninstrumented JVM, and the total number of variable accesses (fields+array indices) performed at runtime. Each column for an algorithm presents, for each benchmark, the total execution time and the slowdown ratio of the program with instrumentation. The time values are given in seconds. The slowdown ratio is the ratio of the difference between the instrumented runtime and the uninstrumented runtime to the uninstrumented runtime. The number of variable accesses checked for races is important for assessing

[1] `Raja` can be obtained at *http://raja.sourceforge.net/*.
[2] `Scimark` can be obtained at *http://math.nist.gov/scimark2/*.

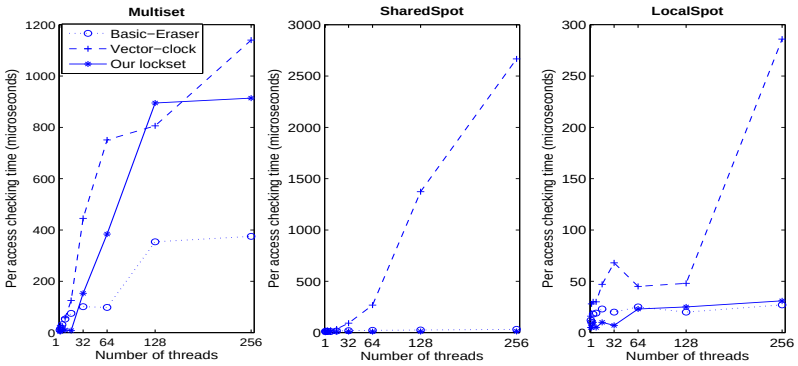

Figure 4: Per access race checking time against the increasing number of threads

the amount of work carried out by the algorithm during execution and average checking time for each variable access.

Table 2 lists the results of our experiments with Goldilocks where checks for fields on which a race is detected are disabled. This is a more realistic setting to judge the overhead of our algorithm in absolute terms. The measurements reported in the first three rows are the same as the ones in Table 1, taken without disabling any checks. The second three rows give the runtime statistics when we followed the approach described above.

4.1 Discussion

The plots in Figure 4 show per access checking times of the three algorithms. The very low acceleration in the per access runtime overhead of our algorithm and Eraser in the `SharedSpot` and `LocalSpot` examples is noteworthy. Short circuit checks in our algorithm allow constant time overhead for thread-local variables and variables protected by a unique lock. This makes our algorithm asymptotically better than the vector-clock algorithm.

The runtime statistics in Table 1 indicate that Goldilocks performs better than the vector-clock algorithm for large-scale programs. As the number of checks done for variable accesses are the same, we can conclude that per variable access checking time of our lockset algorithm on average is less than the vector-clock algorithm.

`SciMark`, `moldyn` and `sor` are well-synchronized programs with few races and a simple locking discipline. Thus the short circuit checks mostly succeed and the overhead of the lockset algorithm is low. However, more elaborate synchronization policies in `Raja`, `raytracer` and `montecarlo` caused long runs of the lockset algorithm, thus the slowdown ratio increases. These programs have a relatively high number of races.

The results indicate that our algorithm works as efficiently as Basic-Eraser while Basic-Eraser can not handle all the synchronization policies used in the benchmarks. The main reason for our algorithm performing slightly better in our experiments is the fact that Basic-Eraser does lockset intersections while checking the accesses. Intersection is fundamentally an expensive operation. Our algorithm, on the other hand, requires insertions and lookups, which can be implemented in constant amortized time. Clearly, a more optimized implementation of Eraser would have performed better. The goal of the comparison with Basic-Eraser was to demonstrate that our algorithm does not have significantly more cost than other lockset algorithms.

Disabling checking accesses to fields on which races were detected dramatically decreases the number of accesses to be checked against races, thus the total runtime of the instrumented program. This can be seen from Table 1. For the benchmarks `moldyn`, `raytracer` and `sor`, the differences in the number of accesses point to this effect.

5 Related work

Dynamic race-detection methods do not suffer from false positives as much as static methods do but are not exhaustive. Eraser [14] is a well-known tool for detecting race conditions dynamically by enforcing the locking discipline that every shared variable is protected by a unique lock. It handles object initialization patterns using a state-based approach but can not handle dynamically changing locksets since it only allows a lockset to get smaller. There is much work that refines the Eraser algorithm by improving the state machine it uses and the transitions to reduce the number of false positives. One such refinement is extending the state-based handling of object initialization and making use of object-oriented concepts [16]. Harrow used thread segments to identify the portions of the execution in which objects are accessed concurrently among threads [8]. Another approach is using a basic vector-clock algorithm to capture thread-local accesses to objects and thus eliminates unnecessary and imprecise applications of the Eraser algorithm [18]. Precise lockset algorithms exist for Cilk programs but their use for real programs is still under question [2]. The general algorithm in [2] is quite inefficient while the efficient version of this algorithm requires programs to obey the umbrella locking discipline, which can be violated by race-free programs.

The approaches that check a happens-before relation [5, 13, 15] are based on vector clocks [10], which create a partial order on program statements. Trade [5] uses a precise vector-clock algorithm. Trade is implemented at the Java byte code level and in interpreter mode of JVM as is our algorithm. To reduce the overhead of the vector clocks for programs with a large number of threads, they use reachability information through the threads, which makes Trade more efficient than other similar tools. Schonberg computes for each thread shared variable sets and concurrency lists to capture the set of shared variables between synchronization points of an execution [15]. His algorithm is imprecise for

	Uninstrumented	Vector-clock	Basic-Eraser	Goldilocks
Benchmark # threads	Runtime (sec.) # accesses	Runtime (sec.) Slowdown	Runtime (sec.) Slowdown	Runtime (sec.) Slowdown
`Raja` 3	8.6 5979629	145.1 15.7	105.9 11.1	70.2 7
`SciMark` 7	28.2 3647012	51.3 0.8	46.1 0.6	33.1 0.1
`moldyn` 7	11.2 8610585	195 16.3	138.9 11.3	92.8 7.2
`raytracer` 7	1.9 5299350	122.8 63.1	79.8 40.6	50 25.1
`montecarlo` 7	5.7 10491747	243.8 41.4	160 26.8	117.5 19.4
`sor` 7	27.2 7696597	145.9 4.3	157.5 4.7	107 2.9

Table 1: Runtime statistics of the benchmark programs

Algorithm	`Raja`	`SciMark`	`moldyn`	`raytracer`	`montecarlo`	`sor`
Runtime	70.2	33.1	92.8	50	117.6	107
Slowdown	7	0.1	7.2	25.1	19.4	2.9
# checks	5979629	3647012	8610585	5299350	10491747	7696597
Runtime*	65.8	35.5	57.0	17.6	111.2	63.8
Slowdown*	6.5	0.2	4	8.2	18.3	1.3
# checks*	5979629	4104754	5268021	1884836	10484544	3416928

* Results after disabling checks to the fields.

Table 2: Runtime statistics when fields with races detected on them are disabled

synchronization disciplines that use locks and needs to be extended for asynchronous coordination to get precision for these disciplines.

Hybrid techniques [12, 18] combine lockset and happens-before analysis. For example, RaceTrack's happens-before computation is based on both vector clocks and locksets. but is not sound as its lockset part of the algorithm is based on Eraser algorithm. Our technique, for the first time, computes a precise happens-before relation using an implementation that makes use of only locksets. Choi et.al. present an unsound runtime algorithm [4] for race detection. They used a static method [3] to eliminate unnecessary checks for well-protected variables. This is a capability we intend to integrate into Goldilocks in the future.

6 Conclusions

In this paper, we present a new sound and precise race-detection algorithm. Goldilocks is based solely on the concept of locksets and is able to capture all mutual-exclusion synchronization idioms uniformly with one mechanism. The algorithm can be used, both in the static or the dynamic context, to develop analyses for concurrent programs, particularly those for detecting data-races, atomicity violations, and failures of safety specifications. In our future work, we plan to develop and integrate into Goldilocks a static analysis technique to reduce the cost of runtime checking.

Acknowledgements

We thank Madan Musuvathi for many interesting discussions that contributed to the implementation technique described in Section 3.

References

[1] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages and Applications*, pages 211–230. ACM, 2002.

[2] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*, pages 298–309, Puerto Vallarta, Mexico, June 28–July 2 1998.

[3] J.-D. Choi, A. Loginov, and V. Sarkar. Static datarace analysis for multithreaded object-oriented programs. Technical Report RC22146, IBM Research, 2001.

[4] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI 02: Programming Language Design and Implementation*, pages 258–269. ACM, 2002.

[5] Mark Christiaens and Koen De Bosschere. Trade, a topological approach to on-the-fly race detection in Java programs. In *JVM 01: Java Virtual Machine Research and Technology Symposium*, pages 105–116. USENIX, 2001.

[6] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Language Design and Implementation*, pages 219–232. ACM, 2000.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL 05: Principles of Programming Languages*, pages 110–121. ACM Press, 2005.

[8] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN 00: Workshop on Model Checking and Software Verification*, pages 331–342. Springer-Verlag, 2000.

[9] Jeremy Manson, William Pugh, and Sarita Adve. The Java memory model. In *POPL 05: Principles of*

Programming Languages, pages 378–391. ACM Press, 2005.

[10] Friedemann Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[11] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[12] E. Pozniansky and A. Schuster. Efficient on-the-fly race detection in multithreaded c++ programs. In *PPoPP 03: Principles and Practice of Parallel Programming*, pages 179–190. ACM, 2003.

[13] M. Ronsse and K. De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, 1999.

[14] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[15] Edith Schonberg. On-the-fly detection of access anomalies. In *PLDI 89: Programming Language Design and Implementation*, pages 313–327, 1989.

[16] Christoph von Praun and Thomas R. Gross. Object race detection. In *OOPSLA 01: Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82. ACM, 2001.

[17] T. Wilkinson. Kaffe: A JIT and interpreting virtual machine to run Java code. http://www.transvirtual.com/, 1998.

[18] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP 05: Symposium on Operating Systems Principles*, pages 221–234. ACM, 2005.

A Example: a task queue

The example in this section, for which pseudocode is provided in Figure 5, demonstrates the use of thread locality, dynamically changing locksets, fork and join operations to ensure mutually exclusive access and how our algorithm is able to uniformly capture all of these idioms. This example consists of a program that schedules tasks (represented by class `Task`) into a queue named `tQ`, dequeues and executes them one by one. Each `Task` instance contains an array `subTasks` of subtasks. Each subtask is a `SubTask` instance. The computation required for a single subtask is represented by a function `Perform` that takes a `SubTask` and produces an integer output. The sum of all the outputs are the final result of the task and this value is also stored in its `out` field. The `Task` object is protected by `Tlock` and the task queue is protected by `Qlock`.

`CreateTask`, given an array `sTs` of subtasks, creates a new task object and enqueues it in the task queue. `PerformNextTask` dequeues a task from `tQ` and calls `ParallelTaskHelper`, which actually performs the task. `ParallelTaskHelper` forks for each subtask a new thread that runs `PerformSubTask`. `PerformSubTask` computes the partial result for the given subtask and adds it to the final result of the task.

Consider the following interleaving of actions during a scenario which begins with creation of two threads `T1` and `T2`:

1. A thread `T1`, by running `CreateTask` with an array containing two subtasks `st0` and `st1` as input,
   (a) creates a new task `oneTask` by calling the `Task` constructor (line 1),
   (b) acquires `Qlock`, calls `tQ.Enqueue (oneTask)`, releases `Qlock` (lines 2-4).
2. A second thread `T2`, runs `PerformNextTask` which
   (a) acquires `Qlock` and calls `tQ.Dequeue ()` that returns `oneTask` (lines 1-3),
   (b) calls `ParallelTaskHelper (oneTask)` (line 4), which creates two threads `T_st0` and `T_st1`, to handle `st0` and `st1` respectively (lines 1-2).
3. `T_st0`, by running `PerformSubTask (oneTask, 0)`,
   (a) calls `Perform ()` (line 1), acquires `Tlock`, and
   (b) adds `subTaskResult` to `oneTask.out`, releases `Tlock` (lines 2-4).
4. The second thread `T_st1`, by running `PerformSubTask (oneTask, 1)`,
   (a) calls `Perform (oneTask.subTasks [1])` (line 1), acquires `Tlock`, and
   (b) adds `subTaskResult` to `oneTask.out`, releases `Tlock` (lines 2-4).
5. Thread `T2`, continuing running `ParallelTaskHelper`,
   (a) joins both threads `T_st0` and `T_st1` (lines 3-4),
   (b) prints `oneTask.out`.

Let us focus on the shared variable `oneTask.out`. In the execution described above, there is no race on `oneTask.out` but the lock protecting it changes dynamically. For example, `oneTask.out` is local to `T1` at the beginning and to `T2` at the end of the scenario.

```
class Task {
  SubTask[n] subTasks;
  int out;
  Task(SubTask[] sT) { subTasks = sT; out = 0;}
}
Queue<Task> tQ;

CreateTask(SubTask[] sTs)
1  oneTask = new Task(sTs);
2  acquire(Qlock)
3     tQ.Enqueue(oneTask);
4  release(Qlock);

PerformNextTask()
1  acquire(Qlock)
2     oneTask = tQ.Dequeue();
3  release(Qlock);
4  ParallelTaskHelper(oneTask);

ParallelTaskHelper(oneTask)
 1  foreach (i < n)
 2    children[i] = fork(PerformSubTask,oneTask,i);
 3  foreach (i < n)
 4    join(children[i]);
 5  print(oneTask.out);

PerformSubTask(oneTask, i)
1  subTaskResult = Perform(oneTask.subTasks[i]);
2  acquire(Tlock);
3     oneTask.out += subTaskResult;
4  release(Tlock);
```

Figure 5: Pseudocode for the task queue example.

We now show how our lockset algorithm handles this execution. Each item below explains how $LS(\texttt{oneTask.out})$ changes after each action during the scenario.

$LS(\texttt{oneTask.out})$ is initially undefined. Our algorithm handles thread-locality by treating thread identifiers similar to locks, and allowing LS to contain thread identifiers.

1. (a) In the constructor of `Task`, `oneTask.out` is first accessed by `T1`. At this point the algorithm sets $LS(\texttt{oneTask.out}) = \{\texttt{T1}\}$.
   (b) After `T1` releases `Qlock`, the rule for release actions adds `Qlock` to the lockset, which yields $LS(\texttt{oneTask.out}) = \{\texttt{T1}, \texttt{Qlock}\}$.
2. (a) After `T2` acquires `Qlock`, since $\texttt{Qlock} \in LS(\texttt{oneTask.out})$ the rule for acquire actions adds `T2` to the lockset so that $LS(\texttt{oneTask.out}) = \{\texttt{T1}, \texttt{Qlock}, \texttt{T2}\}$.
   (b) After `T2` forks `T_st0`, since $\texttt{T2} \in LS(\texttt{oneTask.out})$, `T_st0` gets added to the lockset to yield $LS(\texttt{oneTask.out}) = \{\texttt{T1}, \texttt{Qlock}, \texttt{T2}, \texttt{T\_st0}\}$.
   The same update applies when `T2` creates `T_st1` such that
   $LS(\texttt{oneTask.out}) = \{\texttt{T1}, \texttt{Qlock}, \texttt{T2}, \texttt{T\_st0}, \texttt{T\_st1}\}$.
3. (a) After `T_st0` acquires `Tlock`, since $\texttt{Tlock} \in LS(\texttt{oneTask.out})$, `T_st0` gets added to the lockset which leaves the lockset unchanged.
   (b) After `oneTask.out` is written we check whether $\texttt{T\_st0} \in LS(\texttt{oneTask.out})$. Since the check succeeds, no race is declared and the lockset becomes $LS(\texttt{oneTask.out}) = \{\texttt{T\_st0}\}$. When `T_st0` releases `Tlock`, since $\texttt{T\_st0} \in LS(\texttt{oneTask.out})$ the rule for release actions adds `Tlock` to the lockset to yield $LS(\texttt{oneTask.out}) = \{\texttt{T\_st0}, \texttt{Tlock}\}$.
4. (a) After `T_st1` acquires `Tlock`, since $\texttt{Tlock} \in LS(\texttt{oneTask.out})$, `T_st1` gets added to the lockset, which yields $LS(\texttt{oneTask.out}) = \{\texttt{T\_st0}, \texttt{Tlock}, \texttt{T\_st1}\}$.
   (b) When `oneTask.out` is written, since $\texttt{T\_st1} \in LS(\texttt{oneTask.out})$ before the write action, no race is declared. After the write action we set $LS(\texttt{oneTask.out}) = \{\texttt{T\_st1}\}$. When `T_st1` releases `Tlock`, since $\texttt{T\_st1} \in LS(\texttt{oneTask.out})$ the rule for release actions adds `Tlock` to the lockset to yield $LS(\texttt{oneTask.out}) = \{\texttt{T\_st1}, \texttt{Tlock}\}$.
5. (a) Before `T2` joins `T_st1`, $\texttt{T\_st1} \in LS(\texttt{oneTask.out})$, therefore, after the join action, we add `T2` to the lockset to obtain $LS(\texttt{oneTask.out}) = \{\texttt{T\_st1}, \texttt{Tlock}, \texttt{T2}\}$.
   (b) When `oneTask.out` is accessed by `print`, we check whether $\texttt{T2} \in LS(\texttt{oneTask.out})$. This check succeeds and we do not declare a race. After the access, we set $LS(\texttt{oneTask.out}) = \{\texttt{T2}\}$.

The description above illustrates although $LS(\texttt{oneTask.out})$ shrinks at an access to $LS(\texttt{oneTask.out})$, it can grow whenever a thread executes a release or a fork operation. It is this ability to grow the lockset that is fundamental to capturing dynamic locking idioms.

B Correctness proof of the lockset algorithm

The following lemma defines the conditions that must hold when there is a happens-before edge between a variable access and another action of different threads. This lemma is used in many places throughout the proof of Lemma 2 below. Note that for each state (s_i, LS_i), LS_i is the value of the partial map LS at s_i.

Lemma 1 *Let* $\sigma = (s_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (s_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots (s_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (s_{n+1}, LS_{n+1})$ *be an execution of the program. Let* α_i *in* σ *for some* $i \in [1, n]$ *be the last action that accessed a variable* (o, d).
If $i \xrightarrow{hb} n$ *and* $t_i \neq t_n$ *hold, then there is an action* α_j *such that* $j \in (i, n]$, $i \xrightarrow{hb} j$ *and one of the following conditions holds:*

(a) $\alpha_j = fork(t_n)$, *or*

(b) $t_j = t_n$ *and*

- $\alpha_j = acq(o, l)$ *and* $\exists k \in (i, j).\ \alpha_k = rel(o, l)$ *for a lock* (o, l), *or*
- $\alpha_j = read(o, v)$ *and* $\exists k \in (i, j).\ \alpha_k = write(o, v)$ *for a volatile variable* (o, v), *or*
- $\alpha_j = join(t_i)$.

PROOF. For this proof we will first define "immediate" happens-before (IHB) edges following Definition 1 for the

happens-before relation. There is an IHB edge between p and q, denoted $p \xrightarrow{ihb} q$, only if one of the following conditions holds:

1. $\alpha_p = fork(t_q)$ and α_q is the first action of thread t_q or
2. $\alpha_p = rel(o, l)$ and $\alpha_q = acq(o, l)$ for some lock (o, l) or
3. $\alpha_p = write(o, v)$ and $\alpha_q = read(o, v)$ for some volatile variable (o, v) or
4. α_p is the last action of thread t_p and $\alpha_q = join(t_p)$ or
5. $t_p = t_q$ and $\alpha_p \xrightarrow{po} \alpha_q$ where $\xrightarrow{po}$ is the program order of thread t_p.

Then it is obvious that the happens-before relation is the transitive-closure of the immediate happens-before edges.

Now let $\gamma = i \xrightarrow{ihb} p_1 \xrightarrow{ihb} p_2 \xrightarrow{ihb} \cdots \xrightarrow{ihb} p_u \xrightarrow{ihb} p_{u+1} = n$ be a shortest chain of IHB edges between i and n. Let α_{p_m} be the first action in γ executed by t_n. Then consider the edge $p_{m-1} \xrightarrow{ihb} p_m$ and the types of IHB edges defined above.

- If $p_{m-1} \xrightarrow{ihb} p_m$ is due to condition 1 for IHB edges, then choosing $j = p_{m-1}$ satisfies requirement (a) of the lemma because $i \xrightarrow{hb} p_m$ due to the chain of IHB edges and $\alpha_{p_{m-1}} = fork(t_{p_m})$ must hold.
- If $p_{m-1} \xrightarrow{ihb} p_m$ is due to one of conditions 2-4 for IHB edges, then choosing $j = p_m$ and $k = p_{m-1}$ satisfies requirement (b) of the lemma because $i \xrightarrow{hb} p_m$ due to the chain of IHB edges, $t_{p_m} = t_n$ and α_{p_m} must be an acquire, volatile read or join action.
- $p_{m-1} \xrightarrow{ihb} p_m$ can not be due to condition 5 for IHB edges because we assumed that α_{p_m} is the first action of t_{p_m} in γ, thus $t_{p_{m-1}} \neq t_{p_m}$.

Finally, we conclude that we can always find an action α_j satisfying the lemma as there is at least one shortest chain of IHB edges between i and n. □

The following lemma captures the correctness invariant of our algorithm. It formally characterizes the relationship between the current lockset of each variable and the synchronization operations that occurred in the history of the execution.

Lemma 2 *Let* $\sigma = (s_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (s_2, LS_2) \xrightarrow{\alpha 2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (s_{n+1}, LS_{n+1})$ *be an execution of the program. Let* (o, d) *be a variable that was last accessed by action* α_i *in* σ *for some* $i \in [1, n]$.

(a) Let $l \in Volatile$ *be the field modeling the object lock. Then for all* $x \in Addr$, *we have* $(x, l) \in LS_{n+1}(o, d)$ *iff there exists* j *such that* $j \in (i, n]$, $i \xrightarrow{hb} j$ *and* $\alpha_j = rel(x)$.

(b) Let $v \in Volatile$ *be some volatile field other than the field* l *modeling the object lock. Then for all* $x \in Addr$, *we have* $(x, v) \in LS_{n+1}(o, d)$ *iff there exists* j *such that* $j \in (i, n]$, $i \xrightarrow{hb} j$ *and* $\alpha_j = write(x, v)$.

(c) For all $t \in Tid$, *we have* $t \in LS_{n+1}(o, d)$ *iff there exists* j *such that* $j \in [i, n]$, $i \xrightarrow{hb} j$ *and either* $t_j = t$ *or* $\alpha_j = fork(t)$.

PROOF. We prove the lemma by induction over the length $|\sigma|$ of the execution σ.

Base case: When $|\sigma| = 0$, the claim in the lemma holds trivially because there is no variable (o, d) that is accessed by an action in the execution.

Inductive step: Suppose that the claim in the lemma holds for $(s_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (s_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (s_n, LS_n)$. Consider the transition $(s_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (s_{n+1}, LS_{n+1})$. Let α_i be such that (o, d) was last accessed by α_i where $i \in [1, n]$. The inductive step will consist of proving the lemma for $(s_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (s_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (s_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (s_{n+1}, LS_{n+1})$. We perform a case analysis on α_n.

1. **Local operation:** LS does not change. Therefore the proof follows by a straightforward application of the inductive hypothesis.

2. **Non-volatile variable access:** $\alpha_n = read(o', d')$ or $\alpha_n = write(o', d')$
   Let (o', d') be the variable accessed by α_n. We prove the two cases, $i = n$ and $i \neq n$, separately.

   First, suppose $i = n$, then $(o, d) = (o', d')$. In this case, after α_n is performed, $LS_{n+1}(o, d)$ is set to $\{t_n\}$ even if a race is detected.

   (a) *If direction:* $LS_{n+1}(o, d) = \{t_n\}$ so $\forall (o, l).\ (o, l) \notin LS_{n+1}(o, d)$.
   Only *If direction:* Since $i = n$, there is no action $\alpha_j = rel(o, l)$ such that $j \in (i, n]$.

   (b) *If direction:* $LS_{n+1}(o, d) = \{t_n\}$ so $\forall (o, v).\ (o, v) \notin LS_{n+1}(o, d)$.
   Only *If direction:* Since $i = n$, there is no action $\alpha_j = write(o, v)$ such that $j \in (i, n]$.

   (c) *If direction:* Since $LS_{n+1}(o, d) = \{t_n\}$, $\alpha_i = \alpha_j = \alpha_n$. The happens-before relation is reflexive, so $i \xrightarrow{hb} n$ holds for $i = n$.
   Only if direction: $\alpha_i = \alpha_j = \alpha_n$ and thus $i \xrightarrow{hb} n$ since $t_i = t_j = t_n$. In this case $t_n \in LS_{n+1}(o, d)$ results from the rule for variable access.

   Second, suppose $i \neq n$. Then $(o, d) \neq (o', d')$ and $LS_n(o, d) = LS_{n+1}(o, d)$.

   (a) *If direction:* Let $(o, l) \in LS_{n+1}(o, d)$, then $(o, l) \in LS_n(o, d)$. The inductive hypothesis gives an action $\alpha_j = rel(o, l)$ such that $i \xrightarrow{hb} j$.
   Only if direction: Suppose that there is some action $\alpha_j = rel(o, l)$ such that $i \xrightarrow{hb} j$. As α_n is not a release action, $j \neq n-1$. The inductive hypothesis gives $(o, l) \in LS_n(o, d)$. As $(o, d) \neq (o', d')$, $LS_{n+1}(o, d) = LS_n(o, d)$ so $(o, l) \in LS_{n+1}(o, d)$.

   (b) *If direction:* Let $(o, v) \in LS_{n+1}(o, d)$, then $(o, v) \in LS_n(o, d)$. The inductive hypothesis gives an action $\alpha_j = write(o, v)$ such that $i \xrightarrow{hb} j$.
   Only if direction: Suppose that there is an action $\alpha_j = write(o, v)$ such that $i \xrightarrow{hb} j$ and $j \in (i, n)$. As α_n is not a volatile write action, $j \neq n$. The inductive hypothesis gives $(o, v) \in LS_n(o, d)$. As $(o, d) \neq (o', d')$, $LS_{n+1}(o, d) = LS_n(o, d)$ so $(o, v) \in LS_{n+1}(o, d)$.

(c) *If direction:* Let $t \in LS_{n+1}(o,d)$, then $t \in LS_n(o,d)$. The inductive hypothesis gives an action α_j such that $t_j = t$ or $\alpha_j = fork(t)$ and thus $i \xrightarrow{hb} j$ for $j \in (i,n)$.

Only if direction: Suppose that there is some $j \in [i,n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o,d)$. In this case the rule for variable access guarantees that $t \in LS_{n+1}(o,d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o,d)$. Suppose, aiming to reach a contradiction, that $t_n \notin LS_{n+1}(o,d)$. Because $(o,d) \neq (o',d')$, $t_n \notin LS_n(o,d)$ as $LS_n(o,d) = LS_{n+1}(o,d)$. Because of the inductive hypothesis, there is no $k \in [i, j-1]$ such that $i \xrightarrow{hb} k$, but we have $i \xrightarrow{hb} j = n$. In this case $i \xrightarrow{hb} j = n$ is possible only if there is a direct happens-before edge between α_i and α_n. Because both actions access different variables, this is only possible if $t_i = t_n$. But then it must be true that $t_n \in LS_n(o,d)$ because the variable update rule provides $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change from (s_{i+1}, LS_{i+1}) to (s_n, LS_n). This contradicts with our assumption that $t_n \notin LS_{n+1}(o,d)$. Thus it must be the case that $t_n \in LS_{n+1}(o,d)$.

3. **Lock acquire:** $\alpha_n = acq(o,l)$

(a) *If direction:* Let $(o',l') \in LS_{n+1}(o,d)$ for some lock (o',l'). The rule for acquire guarantees that $(o',l') \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = rel(o',l')$, $j \in (i, n-1]$.

Only if direction: Suppose that there is an action $\alpha_j = rel(o',l')$ such that $i \xrightarrow{hb} j$. As α_n is not a release action, $j \neq n-1$. In this case due to the inductive hypothesis gives $(o',l') \in LS_n(o,d)$. The rule for acquire guarantees that $(o',l') \in LS_{n+1}(o,d)$.

(b) *If direction:* Let $(o,v) \in LS_{n+1}(o,d)$. The rule for acquire guarantees that $(o,v) \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = write(o,v)$, $j \in [i, n-1]$.

Only if direction: Suppose that there is an action $\alpha_j = write(o,v)$ such that $i \xrightarrow{hb} j$. As α_n is not a volatile write action, $j \neq n-1$. In this case the inductive hypothesis gives $(o,v) \in LS_n(o,d)$. The rule for acquire guarantees that $(o,v) \in LS_{n+1}(o,d)$.

(c) *If direction:* Let $t \in LS_{n+1}(o,d)$. Either $t \in LS_n(o,d)$ or $t \notin LS_n(o,d)$. Consider the case of $t \in LS_n(o,d)$. Then the inductive hypothesis gives an action α_j ($j \in (i,n)$) such that $t_j = t$ or $\alpha_j = fork(t)$. Now consider the case when $t \notin LS_n(o,d)$. Then, since $t \in LS_n(o,d)$ gets added by the rule for acquire, it must hold that $t = t_n$ and $(o,l) \in LS_n(o,d)$. In this case the inductive hypothesis gives $j \in [i, n-1]$ such that $\alpha_j = rel(o,l)$ and $i \xrightarrow{hb} j$. As α_j and α_n access the same lock (o,l) $j \xrightarrow{hb} n$. $i \xrightarrow{hb} n$ follows from transitivity of $\xrightarrow{hb}$.

Only if direction: Suppose that there is some $j \in [i,n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o,d)$. In this case the rule for acquire guarantees that $t \in LS_{n+1}(o,d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o,d)$. Then suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o,d)$. The rule for acquire guarantees that $t_n \notin LS_n(o,d)$. It also holds that $(o,l) \notin LS_n(o,d)$ because otherwise t_n would be in $LS_{n+1}(o,d)$. Due to the inductive hypothesis, this implies that there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $t_k = t_n$ or $\alpha_k = fork(t_n)$ or $\alpha_k = rel(o)$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o,d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o,d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$ or $\alpha_k = rel(o,l)$. However, Lemma 1 and the minimality of j imply that $\exists k \in (i,n)$ such that $\alpha_k = rel(o,l)$. Thus it must be the case that $t_n \in LS_{n+1}(o,d)$.

4. **Lock release:** $\alpha_n = rel(o,l)$

(a) *If direction:* Let $(o',l') \in LS_{n+1}(o,d)$ for some lock (o',l'). Suppose $(o',l') \in LS_n(o,d)$. Then the inductive hypothesis gives $i \xrightarrow{hb} n$. Suppose $(o',l') \notin LS_n(o,d)$. Because $(o',l') \in LS_{n+1}(o,d)$, (o',l') is only added if $t_n \in LS_n(o,d)$. In this case due to the inductive hypothesis, there exists $j \in (i, n-1]$ such that $t_j = t_n$ and $i \xrightarrow{hb} j$. Since $t_j = t_n$, $j \xrightarrow{hb} n$ holds and this results in $i \xrightarrow{hb} n$ from the transitive closure.

Only if direction: Suppose that there exists $j \in (i,n]$ such that $\alpha_j = rel(o',l')$ and $i \xrightarrow{hb} j$. Either $(o,l) = (o',l')$ or $(o,l) \neq (o',l')$. Consider the case when $(o,l) \neq (o',l')$. Then $(o',l') \in LS_n(o,d)$ and $j < n$ by the inductive hypothesis. Because of the rule for release, $(o',l') \in LS_{n+1}(o,d)$.

Now consider the case when $(o,l) = (o',l')$. In this case suppose, aiming for a contradiction, that $(o,l) \notin LS_{n+1}(o,d)$. Due to the rule for release, $(o,l) \notin LS_n(o,d)$. It also holds that $t_n \notin LS_n(o,d)$ because otherwise (o,l) would be in $LS_{n+1}(o,d)$. By the inductive hypothesis, this implies that there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $t_k = t_n$ or $\alpha_k = fork(t_n)$

or $\alpha_k = rel(o)$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o,d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o,d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$ or $\alpha_k = rel(o)$. Thus it must be the case that $(o,l) \in LS_{n+1}(o,d)$.

(b) *If direction:* Let $(o,v) \in LS_{n+1}(o,d)$, then $(o,v) \in LS_n(o,d)$. The inductive hypothesis implies what we need. *Only if direction:* Suppose that there exists $j \in [i,n]$ such that $\alpha_j = write(o,v)$ and $i \xrightarrow{hb} j$. In this case $j < n-1$. Then the inductive hypothesis applies and $(o,v) \in LS_n(o,d)$ so $(o,v) \in LS_{n+1}(o,d)$ due to the rule for release.

(c) *If direction:* Let $t \in LS_{n+1}(o,d)$, then $t \in LS_n(o,d)$ because of the rule for release. The inductive hypothesis implies what we want.

Only if direction: Suppose that there is some $j \in [i,n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o,d)$. In this case the rule for release guarantees that $t \in LS_{n+1}(o,d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o,d)$. Suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o,d)$. Thus $t_n \notin LS_n(o,d)$ from the rule for release. By the inductive hypothesis, this implies there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o,d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o,d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. Thus it must be the case that $t_n \in LS_{n+1}(o,d)$.

5. **Volatile read:** $\alpha_n = read(o,v)$ where (o,v) is a volatile variable.

(a) *If direction:* Let $(o,l) \in LS_{n+1}(o,d)$, then $(o,l) \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = rel(o,l)$, $j \in [i,n-1]$.

Only if direction: Suppose that there is an action $\alpha_j = rel(o,l)$ such that $i \xrightarrow{hb} j$. As α_n is not a release action, $j \neq n-1$. In this case the inductive hypothesis gives $(o,l) \in LS_n(o,d)$. Then the rule for volatile read guarantees that $(o,l) \in LS_{n+1}(o,d)$.

(b) *If direction:* Let $(o',v') \in LS_{n+1}(o,d)$ for some volatile variable (o',v'). Then $(o',v') \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = write(o',v')$, $j \in [i,n-1]$.

Only if direction: Suppose that there is an action $\alpha_j = write(o',v')$ such that $i \xrightarrow{hb} j$. As α_n is not a volatile write action, $j \neq n-1$. In this case the inductive hypothesis gives $(o',v') \in LS_n(o,d)$. Then the rule for volatile read guarantees that $(o',v') \in LS_{n+1}(o,d)$.

(c) *If direction:* Let $t \in LS_{n+1}(o,d)$. Either $t \in LS_n(o,d)$ or $t \notin LS_n(o,d)$. Suppose $t \in LS_n(o,d)$. Then the inductive hypothesis holds. Suppose $t \notin LS_n(o,d)$. Then $(o,v) \in LS_n(o,d)$. In this case due to the inductive hypothesis, there exists $j \in [i+1,n-1]$ such that $\alpha_j = write(o,v)$ and $i \xrightarrow{hb} j$. Since $j \xrightarrow{hb} n$, $i \xrightarrow{hb} n$ from the transitive closure.

Only if direction: Suppose that there is some $j \in [i,n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o,d)$. In this case the rule for volatile read guarantees that $t \in LS_{n+1}(o,d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o,d)$. Then suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o,d)$. Thus the rule for volatile read guarantees that $t_n \notin LS_n(o,d)$. It also holds that $(o,v) \notin LS_n(o,d)$ because otherwise t_n would be in $LS_{n+1}(o,d)$. By the inductive hypothesis, this implies there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$ or $\alpha_k = write(o,v)$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o,d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o,d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$ or $\alpha_k = write(o,v)$. However, Lemma 1 and the minimality of j imply that $\exists k \in (i,n)$ such that $\alpha_k = write(o,v)$. Thus it must be the case that $t_n \in LS_{n+1}(o,d)$.

6. **Volatile write:** $\alpha_n = write(o,v)$ where (o,v) is a volatile variable.

(a) *If direction:* Let $(o,l) \in LS_{n+1}(o,d)$, then $(o,l) \in LS_n(o,d)$. The inductive hypothesis implies what we need. *Only if direction:* Suppose that there exists $j \in [i,n]$ such that $\alpha_j = rel(o,l)$ and $i \xrightarrow{hb} j$. In this case $j < n-1$. Then the inductive hypothesis applies and $(o,l) \in LS_n(o,d)$ so $(o,l) \in LS_{n+1}(o,d)$ due to the rule for volatile write.

(b) *If direction:* Let $(o',v') \in LS_{n+1}(o,d)$ for some volatile variable (o',v'). Consider the case when

$(o', v') \in LS_n(o, d)$. Then the inductive hypothesis implies what we need. Now consider the case when $(o', v') \notin LS_n(o, d)$. Then $t_n \in LS_n(o, d)$, since $(o', v') \in LS_{n+1}(o, d)$ as a result of the rule for volatile writes. In this case due to the inductive hypothesis, there exists $j \in (i, n-1]$ such that $t_j = t_n$ or $\alpha_j = fork(t_n)$ and $i \xrightarrow{hb} j$. Since $j \xrightarrow{hb} n$ as they are both executed by t_n, $i \xrightarrow{hb} n$ from transitivity of $\xrightarrow{hb}$.

Only if direction: Suppose that there exists $j \in [i, n]$ such that $\alpha_j = write(o', v')$ and $i \xrightarrow{hb} j$. Either $(o, v) = (o', v')$ or $(o, v) \neq (o', v')$. Consider the case when $(o, v) \neq (o', v')$. Then $(o', v') \in LS_n(o, d)$. Due to the rule for volatile write, $(o', v') \in LS_{n+1}(o, d)$.

Now consider the case when $(o, v) = (o', v')$. In this case suppose, aiming for a contradiction, that $(o, v) \notin LS_{n+1}(o, d)$. Due to the rule for volatile write, $(o, v) \notin LS_n(o, d)$. It also holds that $t_n \notin LS_n(o, d)$ because otherwise (o, v) would be in $LS_{n+1}(o, d)$. By the inductive hypothesis, this implies that there is no $k \in [i, n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = write(o, v)$ or $t_k = t_n$ or $\alpha_k = fork(t_n)$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o, d)$ and $LS(o, d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o, d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o, d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$ or $\alpha_k = write(o, v)$. Thus it must be the case that $(o, v) \in LS_{n+1}(o, d)$.

(c) *If direction:* Let $t \in LS_{n+1}(o, d)$, then $t \in LS_n(o, d)$. The inductive hypothesis implies what we need.

Only if direction: Suppose that there is some $j \in [i, n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o, d)$. In this case the rule for volatile write guarantees that $t \in LS_{n+1}(o, d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o, d)$. Suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o, d)$. Thus $t_n \notin LS_n(o, d)$ from the rule for volatile write. By the inductive hypothesis, this implies there is no $k \in [i, n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o, d)$ and $LS(o, d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o, d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o, d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. Thus it must be the case that $t_n \in LS_{n+1}(o, d)$.

7. **Thread fork:** $\alpha_n = fork(t)$

(a) *If direction:* Let $(o, l) \in LS_{n+1}(o, d)$. Then the rule for fork guarantees that $(o, l) \in LS_n(o, d)$. The inductive hypothesis gives an action $\alpha_j = rel(o, l)$, $j \in [i, n-1]$.

Only if direction: Suppose that there is an action $\alpha_j = rel(o, l)$ such that $i \xrightarrow{hb} j$. As α_n is not a release action, $j \neq n-1$. In this case due to the inductive hypothesis $(o, l) \in LS_n(o, d)$. The rule for fork guarantees that $(o, l) \in LS_{n+1}(o, d)$.

(b) *If direction:* Let $(o, v) \in LS_{n+1}(o, d)$, then $(o, v) \in LS_n(o, d)$. The inductive hypothesis gives an action $\alpha_j = write(o, v)$, $j \in [i, n-1]$.

Only if direction: Suppose that there exists $j \in [i, n]$ such that $\alpha_j = write(o, v)$ and $i \xrightarrow{hb} j$. As α_n is not a volatile write action, $j \neq n-1$. In this case due to the inductive hypothesis $(o, v) \in LS_n(o, d)$. The rule for fork guarantees that $(o, v) \in LS_{n+1}(o, d)$.

(c) *If direction:* Let $t' \in LS_{n+1}(o, d)$. There are two cases: $t' \in LS_n(o, d)$ or $t' \notin LS_n(o, d)$ Consider the case $t' \in LS_n(o, d)$. Then the inductive hypothesis gives an action α_j such that $t_j = t'$ or $\alpha_j = fork(t')$. Now consider the case $t' \notin LS_n(o, d)$. Then the forked thread is t', namely $t = t'$. Then $\alpha_j = \alpha_n$, there is a fork of t' by t_n. This means the rule for fork was applied to add t' to $LS_{n+1}(o, d)$ so $t_n \in LS_n(o, d)$. The inductive hypothesis gives us some $k \in [i, n-1]$ such that $i \xrightarrow{hb} k$ and $t_k = t_n$ or $\alpha_k = fork(t_n)$. And from transitivity of $i \xrightarrow{hb} k$ and $k \xrightarrow{hb} n$ (α_k and α_n are by the same thread), $i \xrightarrow{hb} n$ holds.

Only if direction: Suppose that there is some $j \in [i, n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o, d)$. In this case the rule for fork guarantees that $t \in LS_{n+1}(o, d)$.

Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o, d)$. Then suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o, d)$. Due to the rule for fork, $t_n \notin LS_n(o, d)$. Due to the inductive hypothesis, this implies there is no $k \in [i, n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o, d)$ and $LS(o, d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o, d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o, d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. Thus it must be the case that $t_n \in LS_{n+1}(o, d)$.

8. **Thread join:** $\alpha_n = join(t)$

(a) *If direction:* Let $(o,l) \in LS_{n+1}(o,d)$, then $(o,l) \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = rel(o,l)$, $j \in [i, n-1]$.
Only if direction: Suppose that there is an action $\alpha_j = rel(o,l)$ such that $i \xrightarrow{hb} j$. As α_n is not a release action, $j \neq n-1$. In this case due to the inductive hypothesis $(o,l) \in LS_n(o,d)$. The rule for join guarantees that $(o,l) \in LS_{n+1}(o,d)$.

(b) *If direction:* Let $(o,v) \in LS_{n+1}(o,d)$, then $(o,v) \in LS_n(o,d)$. The inductive hypothesis gives an action $\alpha_j = write(o,v)$, $j \in [i, n-1]$.
Only if direction: Suppose that there exists $j \in [i,n]$ such that $\alpha_j = write(o,v)$ and $i \xrightarrow{hb} j$. As α_n is not a volatile write action, $j \neq n-1$. In this case due to the inductive hypothesis $(o,v) \in LS_n(o,d)$. The rule for join guarantees that $(o,v) \in LS_{n+1}(o,d)$.

(c) *If direction:* Let $t' \in LS_{n+1}(o,d)$. There are two cases: $t' \in LS_n(o,d)$ or $t' \notin LS_n(o,d)$ Consider the case $t' \in LS_n(o,d)$. Then the inductive hypothesis gives an action α_j such that $t_j = t'$ or $\alpha_j = fork(t')$ and $i \xrightarrow{hb} j$. Now consider the case $t' \notin LS_n(o,d)$. Then due to the rule for join $t' = t_n$ and $t \in LS_n(o,d)$ hold. The inductive hypothesis gives us some $k \in [i, n-1]$ such that $i \xrightarrow{hb} k$ and $t_k = t_n$ or $\alpha_k = fork(t_n)$. And from transitivity of $i \xrightarrow{hb} k$ and $k \xrightarrow{hb} n$ (between an action by t and $join(t)$), $i \xrightarrow{hb} n$ holds.
Only if direction: Suppose that there is some $j \in [i,n]$ such that $i \xrightarrow{hb} j$, $t_j = t$ or $\alpha_j = fork(t)$. We will prove this direction for the smallest such j. Either $j = n$ or $j \neq n$ holds. Consider the case when $j \neq n$. Then due to the inductive hypothesis $t \in LS_n(o,d)$. In this case the rule for join guarantees that $t \in LS_{n+1}(o,d)$.
Now consider the case $j = n$. In this case $t = t_n$, thus we will prove that $t_n \in LS_{n+1}(o,d)$. Then suppose, aiming for a contradiction, that $t_n \notin LS_{n+1}(o,d)$. Due to the rule for join, $t_n \notin LS_n(o,d)$. It also holds that $t \notin LS_n(o,d)$ because otherwise t_n would be in $LS_{n+1}(o,d)$. By the inductive hypothesis, these imply that there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$, and 2)there is no $k \in [i,n)$ such that $i \xrightarrow{hb} k$, and $\alpha_k = fork(t)$ or $t_k = t$. The latter implies $t \neq t_i$. If $t_i = t_n$, because $t_i \in LS_{i+1}(o,d)$ and $LS(o,d)$ does not change between (s_{i+1}, LS_{i+1}) and (s_n, LS_n), it results in $t_i = t_n \in LS_n(o,d)$ contradicting our assumption that $t_n \notin LS_{n+1}(o,d)$. If $t_i \neq t_n$, the happens-before edge $i \xrightarrow{hb} n$ contradicts with Lemma 1 and the minimality of j because, as stated above, the inductive hypothesis implies that there is no $i \xrightarrow{hb} k$, and $\alpha_k = fork(t_n)$ or $t_k = t_n$. $i \xrightarrow{hb} n$ also contradicts with the last condition of Lemma 1 because $t \neq t_i$ thus $\alpha_n \neq join(t_i)$. Thus it must be the case that $t_n \in LS_{n+1}(o,d)$.

□

Theorem 1 (Correctness). *Consider a program execution* $\sigma = (s_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (s_2, LS_2) \cdots (s_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (s_{n+1}, LS_{n+1})$. *Let* (o,d) *be a data variable and* $i \in [1, n-1]$ *be such that* α_i *and* α_n *access* (o,d) *but* α_j *does not access* (o,d) *for all* $j \in [i+1, n-1]$. *Then* $t_n \in LS_n(o,d)$ *iff* $i \xrightarrow{hb} n$.

PROOF.

If direction: Suppose that $t_n \in LS_n(o,d)$. Because of Lemma 2 there is an action α_j ($j \in [i,n]$) such that $t_j = t_n$ or $\alpha_j = fork(t_n)$, and $i \xrightarrow{hb} j$. Both of the cases $t_j = t_n$ and $\alpha_j = fork(t_n)$ imply $j \xrightarrow{hb} n$. $i \xrightarrow{hb} n$ follows from the transitivity of $\xrightarrow{hb}$.

Only if direction: Suppose that $i \xrightarrow{hb} n$. Aiming for a contradiction suppose that $t_n \notin LS_n(o,d)$. There are two cases $t_i = t_n$ and $t_i \neq t_n$.

Consider the first case $t_i = t_n$. Because of the fact that $t_i \in LS_{i+1}(o,d)$ and no rule removes t_i from the lockset until the next access to (o,d), $t_n \in LS_n(o,d)$ holds at state s_n, causing a contradiction with our assumption $t_n \notin LS_n(o,d)$.

Now consider the other case $t_i \neq t_n$. Then Lemma 1 implies that there exists an action α_j ($j \in (i,n]$) such that $i \xrightarrow{hb} j$ and $j \xrightarrow{hb} n$. Because we assumed that $t_n \notin LS_n(o,d)$ α_j can not be a fork ($\alpha_j \neq fork(t_n)$) because of Lemma 2. Thus either $\alpha_j = acq(o,l)$ for some lock (o,l) or $\alpha_j = read(o,v)$ for some volatile variable (o,v) or $\alpha_j = join(t_i)$. In all the cases it must hold that $t_j = t_n$ because of Lemma 1. In addition, Lemma 2 implies that $t_j \in LS_{j+1}(o,d)$ because of the edge $i \xrightarrow{hb} j$. Therefore we can conclude that $t_n = t_j \in LS_n(o,d)$ because no rule removes t_j from $LS(o,d)$ until the next access to (o,d). This contradicts with our assumption $t_n \notin LS_n(o,d)$. Thus it must hold that $t_n \in LS_n(o,d)$ if $i \xrightarrow{hb} n$. □