Data Race Detection on Compressed Traces

Anonymous Author(s)

ABSTRACT

We consider the problem of detecting data races in program traces that have been compressed using straight line programs (SLP), which are special context-free grammars that generate exactly one string, namely the trace that they represent. We consider two classical approaches to race detection — using the happens-before relation and the lockset discipline. We present algorithms for both these methods that run in time that is linear in the size of the compressed, SLP representation. Typical program executions almost always exhibit patterns that lead to significant compression. Thus, our algorithms are expected to result in large speedups when compared with analyzing the uncompressed trace. Our experimental evaluation of these new algorithms on standard benchmarks confirms this observation.

ACM Reference Format:
Anonymous Author(s). 2018. Data Race Detection on Compressed Traces. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018).* ACM, New York, NY, USA, 20 pages.

1 INTRODUCTION

Dynamic analysis of multi-threaded programs is the problem of discovering anomalies in a program by examining a single or multiple traces of a program. While dynamic analysis is sometimes performed online as the program is running, it is often performed offline, on a stored trace of the program. The reasons for performing offline dynamic analysis are many. The overhead of analyzing the trace as the program is running maybe large, causing undesirable slowdowns. This is especially true for expensive dynamic analysis techniques which employ heavy-weight machinery such as the use of SMT solvers [22, 45], graph based analysis [5, 20] or even vector clocks [18, 40]. Often, it is desirable to perform multiple, different analyses on a single trace, and the kinds of analyses to be performed may even be unknown at the time the program is being observed. Finally, storing the trace and later replaying it in a controlled environment, can help in debugging programs, in understanding performance overheads and in performance tuning. Trace-driven simulations are used widely in computer architecture for quantitative evaluations of new ideas and designs [24, 34].

However program traces are often huge, recording millions and billions of events. In debugging a large software application, long traces are often necessary to ensure adequate code coverage. This is especially acute for multi-threaded programs where subtle concurrency bugs are often revealed only under specific thread schedules. Therefore useful traces are those that exercise the same program fragment multiple times, under different scenarios; this is substantiated by the observation that some concurrency bugs only manifest themselves in traces with millions of events [12]. In such circumstances, the only way to alleviate the warehousing needs of storing such traces is to compress them [24, 34].

ESEC/FSE 2018, 4 - 9 November, 2018, Lake Buena Vista, Florida, United States 2018.

```java
public class Test extends Thread{
    static final long ITERS = 1000000000L;
    static int y;
    public void inc() {
        y++;
    }
    @Override
    public void run() {
        for (long i = 0; i < ITERS; i++) {
            inc();
        }
    }
    public static void main(String args[]) throws Exception {
        final Test t1 = new Test();
        final Test t2 = new Test();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("y (actual) = " + y);
        System.out.println("y (expected) = " + ITERS*2);
    }
}
```

Figure 1: A simple concurrent program in Java

In this paper, we study the problem of detecting data races in programs by examining compressed traces. Data races are the most common symptom of a programming error in concurrent programming. The naïve approach to solving this problem would be to uncompress the trace and then process it using any one of the many algorithms that have been developed for dynamic data race detection [16, 18, 22, 27, 33, 40, 46]. But is this necessary? Is this naïve algorithm, asymptotically, the best one can hope for? Studying the complexity of problems where the input is represented succinctly has a long history. Starting from the seminal paper by Galperin and Wigderson [21], where they studied the complexity of graph problems when the input graph is represented by a circuit, it has been observed that typically there is an exponential blowup in the complexity of problems when they are solved on compressed inputs [4, 9, 14, 21, 32, 39, 51]. Thus, often the naïve algorithm is the best algorithm asymptotically.

Our results in this paper, fortunately, are the exception to the above rule. We consider two classical race detection approaches — a sound [1] method based on computing Lamport's happens-before relation [29], and the lightweight lockset-based algorithm of Eraser [46] — and extend them to work directly on the compressed trace without first uncompressing it. Our algorithms run in time that is linear in the size of the compressed trace. Thus, we show that compression can in fact be used as an algorithmic principle to speedup the analysis in this context.

To understand why compression actually speeds up the analysis, consider the simple program shown in Figure 1. A single execution of this program generates about 680 million events, taking about 1.3 GB space. However, when this trace is compressed using the algorithm Sequitur [2, 37, 38], the compressed representation only occupies about 34 MB of disk space. The reason why the trace could be compressed so effectively is because the program has a simple loop structure that is executed multiple times. Thus the program

[1] We say a race detector is sound if it never issues any warning on race-free programs or executions. This is often referred to as *precise* [18] in the race detection literature.

trace has a "regular" structure that the compression algorithm exploits. An algorithm processing the uncompressed trace is agnostic to this regularity, and is forced to repeat the same analysis each time the sub-trace corresponding to the loop body in encountered. Compression makes this regular structure "visible", and an algorithm working on the compressed representation can exploit it by only performing an analysis once for each distinct sub-trace.

We consider compression schemes that compress traces by a straight line program (SLP), which is a special context-free grammar where the language of the grammar consists of a single string, namely, the trace being compressed. Several lossless compression schemes, like run-length encoding and the family of Lempel-Ziv encodings [59], can be converted efficiently to SLPs of similar size. Our algorithms on SLPs proceed inductively on the structure of the grammar, and compute, in a compositional fashion, bookkeeping information for each non-terminal in the grammar. Thus, a sub-trace generated by a non-terminal that may appear in many positions in the uncompressed trace, is analyzed only once. For happens-before-based race detection, our algorithm is inspired by the Goldilocks method [10], where happens-before information of events is captured by a set of threads and locks.

We have implemented our algorithms in a tool called ZipTrack. The traces are compressed using a popular SLP-based compresssion algorithm called Sequitur [2]. Our experiments on standard benchmark examples reveal that the algorithms on compressed traces perform well, and on large traces, often have an order of magnitude improvement over algorithms running in the uncompressed setting.

The rest of the paper is organized as follows. After discussing closely related work, we introduce basic notation and classical race detection algorithms in Section 2. In Section 3, we present our happens-before data race algorithm on compressed traces. Our algorithm for checking violations of the lockset discipline on compressed traces is in Section 4. We present our experimental results in Section 5.

Related Work. Type systems to prevent data races have been developed [3, 7, 17]. Since the race detection problem is undecidable, the several static analysis techniques [11, 35, 36, 41, 43, 52, 55, 58] suffer from two problems — they don't scale and they raise many false alarms since they are forced to be conservative. Dynamic race detection techniques can be classified into three categories. There are the unsound lockset-based techniques, which may raise false alarms [46]. Techniques like random testing [47] and static escape analysis [42] can reduce the false alarms in such algorithms, but not eliminate them. The second category of dynamic analysis techniques are predictive runtime analysis techniques [22, 23, 31, 45, 53], where the race detector explores all possible reorderings of the given trace to search for a possible witness of a data race. Since the number of interleavings of a given trace is very large, these do not scale to long traces. The last category of dynamic race detection algorithms are those based on identifying a partial order on the events of a trace, and then searching for a pair of conflicting data accesses that are unordered by the partial order. These techniques are sound and scale to long traces since they typically run in linear time. The simplest, and most commonly used partial order is happens-before [29]. Early vector-clock based algorithms to compute happens-before on traces [16, 33] have been subsequently optimized [18, 40]. A lockset-based method for computing the happens-before partial order was proposed in [10]. Structured parallelism has been exploited to optimize the memory overhead in detecting happens before [8, 15, 44, 50, 57]. More recently, partial order that are weaker that happens before have been proposed for detecting data races, including causal precedence [48] and weak causal precedence [27]. Sofya [28] and RoadRunner [19] are tools that provide a framework for implementing dynamic analysis tools.

2 PRELIMINARIES

In the section we introduce basic notation, our assumptions about concurrent programs, the happens before ordering on events, and some classical algorithms for race detection.

Traces. We will analyze traces of concurrent programs synchronizing through locks while accessing shared memory locations (also referred to as global variables, or simply, variables). Traces are (finite) sequences of events of the form $\langle t : o \rangle$, where t is the thread performing the operation o [2] Operations can be one of the following: forking of a new child thread ($\texttt{fork}(t)$); joining of a child thread ($\texttt{join}(t)$); acquiring and releasing a lock ($\texttt{acq}(\ell)$ and $\texttt{rel}(\ell)$); and, reading and writing to a variable ($\texttt{r}(x)$ and $\texttt{w}(x)$). We will assume that all locks are *reentrant*. That is, a thread t may acquire a lock ℓ multiple times, as long as t holds ℓ. However, t must release ℓ, as many times as it was acquired, before ℓ becomes available for being acquired by some other thread. Therefore, with every release event $e = \langle t : \texttt{rel}(\ell) \rangle$, we can associate a unique acquire event $e' = \langle t : \texttt{acq}(\ell) \rangle$, which is the last $\texttt{acq}(\ell)$-event in thread t before e that is not matched with any $\texttt{rel}(\ell)$ event in thread t before e. This $\texttt{acq}(\ell)$ event e' is said to be the matching acquire of e, and is denoted by $\texttt{match}(e)$. Similarly, for an acquire event e' such that $e' = \texttt{match}(e)$, we will say that e is the matching release of e', and we will also denote this by $\texttt{match}(e')$. For a trace σ, $\sigma \restriction_t$ will denote the subsequence of events performed by thread t.

Notation. Let us fix a trace σ. For an event e, we will say $e \in \sigma$ to denote the fact that e appears in the sequence σ. The set of locks acquired or released in σ will be denoted by $\texttt{Locks}(\sigma)$. $\texttt{Threads}(\sigma)$ will denote the set of threads performing some event in σ; in the presence of forks and joins, this is a bit subtle and we define it as

$$\texttt{Threads}(\sigma) = \{t \mid \exists e \in \sigma.\ e = \langle t : o \rangle \text{ or } e = \langle t' : \texttt{fork}(t) \rangle \text{ or } e = \langle t' : \texttt{join}(t) \rangle\}.$$

For a variable x, the set of $\texttt{w}(x)$-events will be denoted by $\texttt{WEvents}_\sigma(x)$ and the set of $\texttt{r}(x)$-events performed by thread $t \in \texttt{Threads}(\sigma)$ will be denoted by $\texttt{REvents}_\sigma(t, x)$. We will use $\texttt{Rd}(\sigma)$ to denote the set of pairs (t, x) for which $\texttt{REvents}_\sigma(t, x) \neq \emptyset$. Similarly, we will use $\texttt{Wr}(\sigma)$ to denote the set of variables x for which $\texttt{WEvents}_\sigma(x)$ is non-empty. When σ is clear from the context, we may drop it.

For a non-empty subset of events S, we will denote by $\texttt{Last}_\sigma(S)$ the (unique) event $e \in S$, that is latest in σ among the events in S. Similarly, $\texttt{First}_\sigma(S)$ is the event $e \in S$ that is earliest in σ

[2] Formally, each event in a trace is assumed to have a unique event id. Thus, two occurences of a thread performing the same operation will be considered *different* events. Eventhough we will implicitly assume the uniqueness of each event in a trace, to reduce notational overhead, we do not formally introduce event ids.

	Thread 1	Thread 2
1	w(x)	
2	fork(2)	
3		r(x)
4		acq(ℓ)
5		w(y)
6		rel(ℓ)
7	r(x)	
8	acq(ℓ)	
9	rel(ℓ)	
10	w(y)	
11		r(x)
12		acq(ℓ)
13		w(y)
14		rel(ℓ)
15	join(2)	
16	w(y)	

$S \to AB$

$A \to CD$

$C \to EF$

$E \to \langle 1 : \mathtt{w}(x)\rangle\langle 1 : \mathtt{fork}(2)\rangle$

$F \to \langle 2 : \mathtt{r}(x)\rangle\langle 2 : \mathtt{acq}(\ell)\rangle\langle 2 : \mathtt{w}(y)\rangle\langle 2 : \mathtt{rel}(\ell)\rangle$

$D \to \langle 1 : \mathtt{r}(x)\rangle\langle 1 : \mathtt{acq}(\ell)\rangle\langle 1 : \mathtt{rel}(\ell)\rangle\langle 1 : \mathtt{w}(y)\rangle$

$B \to FG$

$G \to \langle 1 : \mathtt{join}(2)\rangle\langle 1 : \mathtt{w}(y)\rangle$

Figure 2: Example trace σ_1 and its SLP representation

amongst the events in S. When S is empty, we say both $\mathsf{First}_\sigma(S)$ and $\mathsf{Last}_\sigma(S)$ are undefined.

Example 2.1. We illustrate the above definitions on the example trace σ_1 shown in Figure 2. We will follow the convention of representing events of a trace from top-to-bottom, where temporally earlier events appear above the later ones. We use e_i to denote the ith event in σ_1. Let $S_1 = \mathsf{REvents}_{\sigma_1}(2, x) = \{e_3, e_{11}\}$ and $S_2 = \mathsf{WEvents}_{\sigma_1}(y) = \{e_5, e_{10}, e_{13}, e_{16}\}$. The set $\mathsf{Rd}(\sigma_1) = \{(1, x), (2, x)\}$ while $\mathsf{Wr}(\sigma_1) = \{x, y\}$. Finally, $\mathsf{Last}_{\sigma_1}(S_1) = e_{11}$, and $\mathsf{First}_{\sigma_1}(S_2) = e_5$.

Orders on Traces. Let us fix a trace σ. If an event e_1 appears earlier in the sequence σ than e_2, then we say e_1 is *trace ordered before* e_2 and denote it as $e_1 <^\sigma_{\mathsf{tr}} e_2$. We say e_1 is *thread ordered before* e_2, denoted by $e_1 <^\sigma_{\mathsf{TO}} e_2$, if e_1 and e_2 are events performed by the same thread and $e_1 <^\sigma_{\mathsf{tr}} e_2$. Our race detection algorithm will rely on computing the *happens before* strict order, which we define next.

Definition 2.2 (Happens Before). Event e in trace σ said to *happen before* event $e' \in \sigma$, denoted $e <^\sigma_{\mathsf{HB}} e'$, if and only if there is a sequence of events $e = e_1, e_2, e_3, \ldots e_n = e'$ such that for every pair (e_i, e_{i+1}) $(i < n)$, $e_i <^\sigma_{\mathsf{tr}} e_{i+1}$ and **one** of the following conditions hold.

(1) $e_i <^\sigma_{\mathsf{TO}} e_{i+1}$,
(2) $e_i = \langle t : \mathtt{rel}(\ell)\rangle$ and $e_{i+1} = \langle t' : \mathtt{acq}(\ell)\rangle$ for some t, t', ℓ,
(3) $e_i = \langle t : \mathtt{fork}(t')\rangle$ and $e_{i+1} = \langle t' : o\rangle$ for some t, t', o, or
(4) $e_i = \langle t' : o\rangle$ and $e_{i+1} = \langle t : \mathtt{join}(t')\rangle$ for some t, t', o.

For any $P \in \{\mathsf{tr}, \mathsf{TO}, \mathsf{HB}\}$, $\leq^\sigma_P$ refers to the partial relation $<^\sigma_P \cup =^\sigma$, where $=^\sigma$ denotes the identity relation on the events of σ. When σ is clear from the context we will drop the superscript from these relations; for example, we will use $\leq_{\mathsf{HB}}$ instead of $\leq^\sigma_{\mathsf{HB}}$.

Finally, we say a pair of events e_1, e_2 are *concurrent* (w.r.t. happens before) if neither $e_1 \leq_{\mathsf{HB}} e_2$, nor $e_2 \leq_{\mathsf{HB}} e_1$; we denote this by $e_1 || _{\mathsf{HB}} e_2$.

We now define races identified by the happens-before relation. A pair of events $e_1 = \langle t_1 : \mathsf{a}_1(x)\rangle$ and $e_2 = \langle t_2 : \mathsf{a}_2(x)\rangle$ (for some variable x) is said to be *conflicting*, denoted $e_1 \asymp e_2$, if $t_1 \neq t_2$ and at least one out of a_1 and a_2 is w. A trace σ is said to have a *happens before race* (HB-*race*, for short) if there is a pair of events $e_1, e_2 \in \sigma$ such that $e_1 \asymp e_2$ and $e_1 || _{\mathsf{HB}} e_2$.

Example 2.3. We illustrate the happens before relation through the trace σ_1 in Figure 2. $e_1 \leq_{\mathsf{HB}} e_3$ because e_2 happens before every event in thread 2 since it forks thread 2. Similarly, we can conclude that $e_{13} \leq_{\mathsf{HB}} e_{16}$ because the join event e_{15} is after every event in thread 2. Another interesting pair is $e_5 \leq_{\mathsf{HB}} e_{10}$. This is because e_4, e_5, e_6 and e_8, e_9 are critical sections over the same lock ℓ, and thus, e_6 happens before e_8. Therefore, $e_5 \leq_{\mathsf{TO}} e_6 \leq_{\mathsf{HB}} e_8 \leq_{\mathsf{TO}} e_{10}$. It is useful to pay attention to a couple of concurrent pairs of events. Events e_3 and e_7 are concurrent, but do not constitute an HB-race because e_3 and e_7 being read events are not conflicting. However, there is an HB-race between events e_{10} and e_{13}; they are concurrent and a conflicting pair of events.

The standard FastTrack style vector clock algorithm [16, 18, 29, 33, 40] detects if a given trace has a race and runs in time $O(nT \log n)$ and uses space $O((V + L + T)T \log n)$ for a trace with n events, T threads, L locks and V variables.

Goldilocks Algorithm. Goldilocks algorithm ,[10] is another algorithm that detects the presence of HB-races. In order to formally describe the algorithm, let us first fix some notations. Consider the function After_σ defined as follows:

$$\begin{aligned}\mathsf{After}_\sigma(e) = \quad & \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t : o\rangle.\ e \leq^\sigma_{\mathsf{HB}} e'\} \\ \cup \quad & \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t' : \mathtt{fork}(t)\rangle.\ e \leq^\sigma_{\mathsf{HB}} e'\} \\ \cup \quad & \{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathtt{rel}(\ell)\rangle.\ e \leq^\sigma_{\mathsf{HB}} e'\}\end{aligned}$$

Thus, informally, $\mathsf{After}_\sigma(e)$ is the set of all threads and locks that have an event HB-after e.

Then, for every prefix σ' of the trace, and for every thread t and variable x in σ', the Goldilocks algorithm maintains the set $\mathsf{GLS}^{\mathsf{R}}_{\sigma'}(t, x)$ defined by

$$\mathsf{GLS}^{\mathsf{R}}_{\sigma'}(t, x) = \mathsf{After}_{\sigma'}(\mathsf{Last}_{\sigma'}(\mathsf{REvents}_{\sigma'}(t, x)))$$

and for every variable x in σ', the set

$$\mathsf{GLS}^{\mathsf{W}}_{\sigma'}(x) = \mathsf{After}_{\sigma'}(\mathsf{Last}_{\sigma'}(\mathsf{WEvents}_{\sigma'}(x)))$$

where, $\mathsf{After}_{\sigma'}(\text{undefined})$ is assumed to be the empty set.

Finally, a race is declared after observing an event e such that one of the following hold:

(1) $e = \langle t : \mathtt{w}(x)\rangle$ and either $t \notin \mathsf{GLS}^{\mathsf{W}}_{\sigma'}(x)$ or $t \notin \mathsf{GLS}^{\mathsf{R}}_{\sigma'}(t', x)$ for some thread $t' \in \mathsf{Threads}(\sigma')$
(2) $e = \langle t : \mathtt{r}(x)\rangle$ and $t \notin \mathsf{GLS}^{\mathsf{W}}_{\sigma'}(x)$.

where σ' is the prefix until the event e. This algorithm runs in time $O(n(L+TV))$ and uses space $O(TV(T+L))$ for a trace with n events, T threads, L locks and V variables.

Eraser's Lockset Algorithm. A low overhead technique to detect potential races, is the lockset algorithm [46]. The basic idea here, is to maintain, for every variable x, the set of locks that protect each access to x, and check if this set becomes empty as the execution proceeds. We recall the details of this technique here. We will assume that none of the elements in the set $\mathcal{D} = \{\Lambda\} \cup \{\Lambda_t \mid t \text{ is a thread}\}$ are locks used by the program. The elements of the set $\mathcal{D}$ are "dummy" or fake locks introduced by the algorithm to ensure that alarms are not raised when a (global) variable is only read (and never written to), and when a variable is accessed by only one thread [40]. For a read/write event event $e = \langle t : \mathsf{a}(x)\rangle$ (where a is either r or w) in trace σ, $\mathsf{LocksHeld}_\sigma(e)$ is the set of locks held by t when

	Thread 1	Thread 2
1	r(x)	
2	acq(ℓ)	
3	w(y)	
4	rel(ℓ)	
5		acq(ℓ)
6		r(x)
7		w(y)
8		rel(ℓ)
9		r(x)
10	w(z)	

$S \to UV$

$U \to WX$

$W \to \langle 1 : \mathsf{r}(x)\rangle\langle 1 : \mathsf{acq}(\ell)\rangle$

$X \to \langle 1 : \mathsf{w}(y)\rangle\langle 1 : \mathsf{rel}(\ell)\rangle\langle 2 : \mathsf{acq}(\ell)\rangle$

$V \to YZ$

$Y \to \langle 2 : \mathsf{r}(x)\rangle\langle 2 : \mathsf{w}(y)\rangle$

$Z \to \langle 2 : \mathsf{rel}(\ell)\rangle\langle 2 : \mathsf{r}(x)\rangle\langle 1 : \mathsf{w}(z)\rangle$

Figure 3: Example trace σ_2 and its SLP representation

e is performed. Using this, for an event $e = \langle t : \mathsf{a}(x)\rangle$ we define $\mathsf{LockSet}_\sigma(e)$ to be

$$\mathsf{LockSet}_\sigma(e) = \begin{cases} \{\Lambda, \Lambda_t\} \cup \mathsf{LocksHeld}_\sigma(e) & \text{if } \mathsf{a} = \mathsf{r} \\ \{\Lambda_t\} \cup \mathsf{LocksHeld}_\sigma(e) & \text{if } \mathsf{a} = \mathsf{w} \end{cases}$$

For a variables x and thread t, let $\mathsf{Access}_\sigma(t, x)$ be the set of all events in $\sigma \restriction_t$ whose corresponding operations are either $\mathsf{r}(x)$ or $\mathsf{w}(x)$. Then,

$$\mathsf{LockSet}_\sigma(t, x) = \bigcap_{e \in \mathsf{Access}_\sigma(t,x)} \mathsf{LockSet}_\sigma(e).$$

As per convention, when $\mathsf{Access}_\sigma(t, x) = \emptyset$ (i.e., thread t never accesses the variable x), the right hand side of the above equation is assumed to be $\mathsf{Locks}(\sigma) \cup \mathcal{D}$. A few observations about these definitions are in order. First $\mathsf{LockSet}_\sigma(t, x)$ is always non-empty because $\Lambda_t \in \mathsf{LockSet}_\sigma(t, x)$. Second, if all events in $\mathsf{Access}_\sigma(t, x)$ are read events, then $\Lambda \in \mathsf{LockSet}_\sigma(t, x)$. The lockset discipline is said to be *violated* in trace σ, if for some variable x,

$$\bigcap_{t \in \mathsf{Threads}(\sigma)} \mathsf{LockSet}_\sigma(t, x) = \emptyset.$$

Note that the Eraser algorithm crucially depends upon the accurate computation of $\mathsf{LocksHeld}_\sigma(e)$. For this, we need to record, for each thread t and lock ℓ, the number of times ℓ has been acquired, without being released, which can be maintained using an integer variable.

We conclude this description by highlighting the importance of the locks in $\mathcal{D}$ that were introduced. Take $\mathsf{LS} = \cap_{t \in \mathsf{Threads}(\sigma)} \mathsf{LockSet}_\sigma(t, x)$. If a variable x is only accessed by a single thread t_1, then LS is non-empty because it contains Λ_{t_1}. And if a variable x is only read and never written to, then LS is again non-empty because it contains Λ. The Eraser algorithm [46] checks for violation of the lockset principle by maintaining the lockset for each thread-variable pair. It runs in time $O(n(L + \log r))$ and uses space $O(TL \log r + V(T + L))$ where n, T, L and V are the number of events, threads, locks, and variables respectively, and r is the maximum number of times a thread acquires a lock without releasing it.

Example 2.4. We illustrate the lockset algorithm on a couple of examples. Consider the trace σ_2 in Figure 3. The relevant locksets are as follows.

$$\mathsf{LockSet}_{\sigma_2}(1, x) = \{\Lambda, \Lambda_1\} \quad \mathsf{LockSet}_{\sigma_2}(2, x) = \{\Lambda, \Lambda_2\}$$
$$\mathsf{LockSet}_{\sigma_2}(1, y) = \{\Lambda_1, \ell\} \quad \mathsf{LockSet}_{\sigma_2}(2, y) = \{\Lambda_2, \ell\}$$
$$\mathsf{LockSet}_{\sigma_2}(1, z) = \{\Lambda_1\} \quad \mathsf{LockSet}_{\sigma_2}(2, z) = \{\Lambda, \Lambda_1, \Lambda_2, \ell\}$$

Observe that $\mathsf{LockSet}_{\sigma_2}(2, z)$ is the set of all locks because thread 2 does not access z. The trace σ_2 does not violate the lockset discipline. Informally, the reason for this is because variable x is only read by both threads, accesses to variable y is always protected by lock ℓ, and variable z is local to thread 1. Trace σ_2 also contains no HB-race.

For trace σ_1 from Figure 2,

$$\mathsf{LockSet}_{\sigma_1}(1, x) = \{\Lambda_1\} \quad \mathsf{LockSet}_{\sigma_1}(1, y) = \{\Lambda_1\}$$
$$\mathsf{LockSet}_{\sigma_1}(2, x) = \{\Lambda, \Lambda_2\} \quad \mathsf{LockSet}_{\sigma_1}(2, y) = \{\Lambda_2, \ell\}.$$

The lockset discipline is violated on both variables x and y. On the other hand, there is an HB-race only on variable y (events 10 and 13; see Example 2.3). Thus, the lockset discipline may falsely conclude the presence of races; it is only a lightweight approximate approach.

Straight Line Programs (SLP). We consider traces that are compressed using special context-free grammars called straight line programs (SLP). Recall that a context-free grammar (in Chomsky Normal Form) is $G = (T, N, S, R)$, where T is the set of terminals, N the set of non-terminals, $T \cup N$ is the set of symbols, $S \in N$ is the start symbol, and R is the set of rules in which each rule in R is either $A \to a$ or $A \to BC$, for $A, B, C \in N$ and $a \in T$. A *straight line program* is a context free grammar such that (a) for every non-terminal A, there is exactly one rule where A appears on the left, and (b) the non-terminals are ranked in such that way that in every rule, the non-terminals on the right are of larger rank than the non-terminal on the left of the rule, i.e., for rules $A \to BC$, $A \prec B$ and $A \prec C$. It is easy to observe that the language of the grammar contains a single string, namely, the one that is being succinctly represented by the SLP. Without loss generality, we will assume that every non-terminal in the SLP is *useful*, i.e., every non-terminal in the grammar appears in some sentential form in the unique derivation in the grammar. Thus, the language associated with any non-terminal A has a single string. We will call this (unique) string generated by non-terminal A a *chunk*, and denote it by $[\![A]\!]$. We will often abuse notation and refer to both $[\![A]\!]$ and A as "A". For example, $\mathsf{Locks}(A)$ will mean $\mathsf{Locks}([\![A]\!])$.

The *size* of an SLP $G = (T, N, S, R)$ will be taken to be $|T| + |N|$; note that this measure of size is linearly related to other measures of size one might consider like $|R|$ or sum of the sizes of all the rules in R. We make a couple of observations about the size of an SLP versus the size of the trace it represents. First, every trace $\sigma = e_1, e_2, \ldots e_n$ can be represented by a "trivial" SLP of size $O(n)$ as follows. The non-terminals are $\{A_{[i,i]} \mid 1 \le i \le n\} \cup \{A_{[1,i]} \mid 1 \le i \le n\}$ with start symbol $A_{[1,n]}$. Intuitively, $A_{[i,i]}$ represents the string e_i, while $A_{[1,i]}$ represents the prefix of length i. This is accomplished by the rules — $A_{[i,i]} \to e_i$ and $A_{[1,i]} \to A_{[1,i-1]}A_{[i,i]}$ for each $1 \le i \le n$. Second, the SLP representation of a string σ maybe exponentially smaller than σ itself. For example, take $\sigma = a^{2^n}$. An $O(n)$ SLP representation for σ is as follows: $N = \{A_i \mid 0 \le i \le n\}$ with rules $A_0 \to a$, and $A_{i+1} \to A_i A_i$. One can inductively observe that $[\![A_i]\!] = a^{2^i}$, and so $[\![A_n]\!] = \sigma$.

Example 2.5. Figure 2 describes an SLP representation of trace σ_1. The rules for E, F, D, and G are not strictly in the format of an SLP, but it can easily be converted into one; the representation in Figure 2 is sufficient for our illustrative purposes. We will again use e_i to denote the ith event of σ_1. Chunk E represents e_1, e_2, F

represents e_3, e_4, e_5, e_6 and $e_{11}, \ldots e_{14}$, D represents e_7, e_8, e_9, e_{10}, and G represents e_{15}, e_{16}. The sub-traces represented by the other non-terminals can be similarly discovered. As mentioned before, we will confuse the notation distinguishing between a non-terminal and the string it represents. Thus, for example, $\mathsf{Threads}(E) = \{1, 2\}$.

Similarly, the SLP for σ_2 is shown in Figure 3. The sub-traces represented by non-terminals need not conform to thread and critical section boundaries. For example, the chunk $[\![X]\!]$ has partial critical sections of different threads.

Several well known algorithms for SLP based compression are known in the literature. The most basic and popular one is Sequitur [37, 38]. Sequitur takes a string as an input, and generates an SLP representing the trace. It runs in time and space linear in the size of the input string. The Sequitur algorithm works in an online incremental fashion; it reads the input string one character at a time, and updates the SLP generated so-far. It maintains a list of digrams (symbol pairs) that occur somewhere in the SLP so-far. On seeing a new character, the algorithm appends it at the end of the rule corresponding to the start symbol. The new digram formed (by appending the new character to the last symbol of the rule) is added to the list of digrams, if it is not already present. Otherwise, a new rule, with a fresh nonterminal generating the digram, is added to the SLP, and every occurence of the digram is replaced by the freshly introduced non-terminal. At every step, non-terminals, that are not useful, are also removed. Other popular grammar based compression schemes include Sequential [56], LZ77 [59], LZW [54], Bisection [26], longest match [25] and Re-Pair [30].

3 HB-RACES IN COMPRESSED TRACES

In this section, we will present our algorithm for detecting HB-races in compressed traces represented by SLPs. The algorithm's running time will be linear in the size of the SLP (as opposed to the uncompressed algorithm). While it is very different from the classical vector clock algorithm, it is similar in flavor to the Goldilocks Algorithm.

3.1 Detecting Cross-Races

Our algorithm will proceed inductively. Starting from the non-terminals of largest rank, we will proceed to determine for each non-terminal A, whether there is an HB-race amongst the events in the chunk that A generates. In other words, for each non-terminal A, we will determine the predicate $\mathsf{Race?}(A)$ which is true if and only if there is an HB-race between events in $[\![A]\!]$. For a non-terminal A, whose (only) rule is of the form $A \to a$, where a is an event, $\mathsf{Race?}(A)$ is clearly false, because $[\![A]\!]$, in this case, has only one event.

Let us now consider the case when A has the rule $A \to BC$, where B and C are non-terminals of higher rank. If there is a race in chunk $[\![A]\!]$ between events (say) e and e', then it is one of two kinds. The first case is when e and e' both belong to chunk $[\![B]\!]$ or both belong to chunk $[\![C]\!]$. The existence of such races can be determined by computing (inductively) the predicate $\mathsf{Race?}(B)$ and $\mathsf{Race?}(C)$. The other possibility is that $e \in [\![B]\!]$ while $e' \in [\![C]\!]$. How we discover the presence of such *cross-races*, is the main challenge we need to overcome.

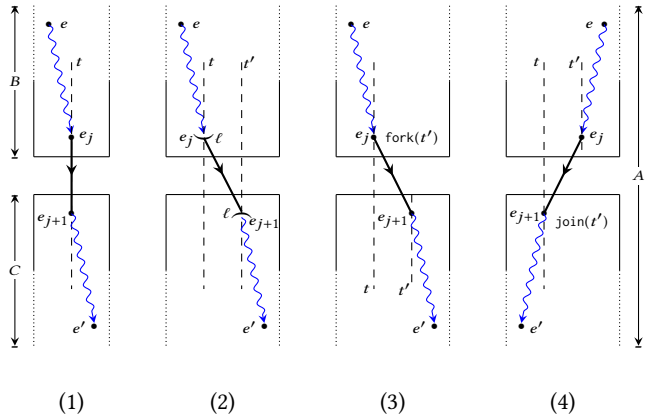

Figure 4: Illustrating the various scenarios that establish $e_j <^{BC}_{\mathsf{HB}} e_{j+1}$. In (2), '$\frown\ \ell$' represents $\mathsf{acq}(\ell)$, '$\smile\ \ell$' is $\mathtt{rel}(\ell)$.

Consider two events e, e' such that $e \in [\![B]\!]$ and $e' \in [\![C]\!]$. Suppose $e \leq^{BC}_{\mathsf{HB}} e'$. Then, there is a sequence $e = e_1, e_2, \ldots e_n = e'$ that satisfies the conditions in Definition 2.2. Thus, for $1 \leq i \leq n-1$, we have the trace order $e_i <^{BC}_{\mathsf{tr}} e_{i+1}$. Also, $e = e_1 \in [\![B]\!]$, and $e_n = e' \in [\![C]\!]$. This means that there exists j such that for all $i \leq j$, $e_i \in [\![B]\!]$, and for all $i \geq j+1$, $e_i \in [\![C]\!]$. In other words, (e_j, e_{j+1}) is how the sequence $e_1, \ldots e_n$ "crosses" the B-C boundary (see Figure 4). Observe that we have $e = e_1 \leq^{B}_{\mathsf{HB}} e_j$ and $e_{j+1} \leq^{C}_{\mathsf{HB}} e_n = e'$. It is important to note that the relationship between e and e_j (and e_{j+1} and e') only depends on the events in chunk $[\![B]\!]$ ($[\![C]\!]$). Depending on which of the conditions (1), (2), (3), and (4) of Definition 2.2 hold for the pair (e_j, e_{j+1}), we have one of the following: either e_j and e_{j+1} are events of the same thread, or e_j is a release event and e_{j+1} is an acquire event on the same lock, or e_j is a fork event and e_{j+1} is an event of the child thread, or e_j is a join event and e_{j+1} is an event of the parent thread. These scenarios are illustrated in Figure 4. Thus, if an event $e \in B$ *happens-before* an event $e' \in C$ then there is a common thread/lock through which the ordering is "communicated" across the B-C boundary. The converse of this observation is also true. We now make this intuition precise.

For a trace σ, and event $e \in \sigma$, recall the function After_σ:

$$\begin{aligned}\mathsf{After}_\sigma(e) = &\{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t : o\rangle.\ e \leq^{\sigma}_{\mathsf{HB}} e'\}\\ &\cup \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t' : \mathtt{fork}(t)\rangle.\ e \leq^{\sigma}_{\mathsf{HB}} e'\}\\ &\cup \{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathtt{rel}(\ell)\rangle.\ e \leq^{\sigma}_{\mathsf{HB}} e'\}\end{aligned}$$

We can, dually, define the set of locks/threads that have an event HB-before e in σ.

$$\begin{aligned}\mathsf{Before}_\sigma(e) = &\{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t : o\rangle.\ e' \leq^{\sigma}_{\mathsf{HB}} e\}\\ &\cup \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t' : \mathtt{join}(t)\rangle.\ e' \leq^{\sigma}_{\mathsf{HB}} e\}\\ &\cup \{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathsf{acq}(\ell)\rangle.\ e' \leq^{\sigma}_{\mathsf{HB}} e\}\end{aligned}$$

The main observation that underlies the algorithm is that After and Before sets can be used to discover HB ordering between events across chunks.

Lemma 3.1. *Consider events $e \in [\![B]\!]$ and $e' \in [\![C]\!]$. $e \leq^{BC}_{\mathsf{HB}} e'$ iff* $\mathsf{After}_B(e) \cap \mathsf{Before}_C(e') \neq \emptyset$.

Lemma 3.1 suggests cross races in chunk BC can be discovered by maintaining the after and before sets of data access events. However, we don't need to maintain these sets for all access events; instead

we can do it only for the first and last events. This is the content of the next lemma.

Lemma 3.2. *If there is no* HB*-race in* $[\![B]\!]$ *or in* $[\![C]\!]$*, and if there is an* HB*-race between events* $e \in [\![B]\!]$ *and* $e' \in [\![C]\!]$ *then, there is an* HB*-race between* last^e_B *and* $\mathsf{first}^{e'}_C$*, where*

$$\mathsf{last}^e_B = \begin{cases} \mathsf{Last}_B(\mathsf{REvents}_B(t,x)) & \text{if } e = \langle t : \mathtt{r}(x)\rangle \\ \mathsf{Last}_B(\mathsf{WEvents}_B(x)) & \text{if } e = \langle t : \mathtt{w}(x)\rangle \end{cases}$$

and

$$\mathsf{first}^{e'}_C = \begin{cases} \mathsf{First}_C(\mathsf{REvents}_C(t,x)) & \text{if } e' = \langle t' : \mathtt{r}(x)\rangle \\ \mathsf{First}_C(\mathsf{WEvents}_C(x)) & \text{if } e' = \langle t' : \mathtt{w}(x)\rangle \end{cases}$$

Lemma 3.2 suggests that in order to check for *cross* races, it is enough to inductively maintain the after sets of the last read/write events and the before sets of the first read/write events of each variable and thread. We will denote these sets by ALRd, ALWr, BFRd and BFWr. Formally,

$$\begin{array}{ll} \mathsf{ALRd}_D(t,x) & = \mathsf{After}_D(\mathsf{Last}_D(\mathsf{REvents}_D(t,x))) \\ \mathsf{ALWr}_D(x) & = \mathsf{After}_D(\mathsf{Last}_D(\mathsf{WEvents}_D(x))) \\ \mathsf{BFRd}_D(t,x) & = \mathsf{Before}_D(\mathsf{First}_D(\mathsf{REvents}_D(t,x))) \\ \mathsf{BFWr}_D(x) & = \mathsf{Before}_D(\mathsf{First}_D(\mathsf{WEvents}_D(x))) \end{array} \tag{1}$$

where we set both $\mathsf{After}_D(\text{undefined})$ and $\mathsf{Before}_D(\text{undefined})$ to be $\emptyset$.

Based on all of these observations we can conclude that for a non-terminal A with rule $A \to BC$, we have,

$$\begin{array}{ll} \mathsf{Race?}(A) = & \mathsf{Race?}(B) \vee \mathsf{Race?}(C) \vee \\ \bigvee_{x \in \mathsf{Wr}(B) \cap \mathsf{Wr}(C)} & \mathsf{ALWr}_B(x) \cap \mathsf{BFWr}_C(x) = \emptyset \\ \bigvee_{x \in \mathsf{Wr}(B), (t,x) \in \mathsf{Rd}(C)} & \mathsf{ALWr}_B(x) \cap \mathsf{BFRd}_C(t,x) = \emptyset \\ \bigvee_{(t,x) \in \mathsf{Rd}(B), x \in \mathsf{Wr}(C)} & \mathsf{ALRd}_B(t,x) \cap \mathsf{BFWr}_C(x) = \emptyset \end{array} \tag{2}$$

Thus, our race detection algorithm will be complete if we can effectively compute $\mathsf{ALRd}_B(t,x)$, $\mathsf{ALWr}_B(x)$, $\mathsf{BFRd}_C(t,x)$, and $\mathsf{BFWr}_C(x)$. We embark on this challenge in the next section. Our definition of the predicate Race? is correct and we state this next.

Theorem 3.3. *For any non-terminal* A*,* $\mathsf{Race?}(A) = \mathsf{true}$ *if and only if there are events* $e_1, e_2 \in [\![A]\!]$ *such that* $e_1 \asymp e_2$ *and* $e_1 \| _{\mathsf{HB}} e_2$*.*

Example 3.4. Let us illustrate the ideas presented in this section through some examples. We will consider traces σ_1 and its SLP in Figure 2, and σ_2 with its SLP in Figure 3.

We begin by giving examples of Before and After sets.

$$\begin{array}{ll} \mathsf{After}_E(e_1) = \{1,2\} & \mathsf{After}_C(e_1) = \{1,2,\ell\} \\ \mathsf{Before}_G(e_{16}) = \{1,2\} & \mathsf{Before}_B(e_{16}) = \{1,2,\ell\} \\ \mathsf{After}_W(e_1) = \{1\} & \mathsf{After}_U(e_1) = \{1,2,\ell\} \\ \mathsf{Before}_X(e_3) = \{1\} & \mathsf{Before}_U(e_3) = \{1,\ell\} \end{array}$$

Let us highlight the significant aspects of these examples. $2 \in \mathsf{After}_E(e_1)$ because of $e_2 = \langle 1 : \mathtt{fork}(2)\rangle$ and $\ell \in \mathsf{After}_C(e_1)$ because of event $e_6 = \langle 2 : \mathtt{rel}(\ell)\rangle$. On the other hand, $\ell \notin \mathsf{After}_W(e_1)$ because there is no $\mathtt{rel}(\ell)$ event in chunk $[\![W]\!]$ (of σ_2). But when considering the chunk $[\![U]\!]$ (of σ_2), we have $\ell \in \mathsf{After}_W(e_1)$ because of the event $e_4 = \langle 1 : \mathtt{rel}(\ell)\rangle$. Next, $2 \in \mathsf{Before}_G(e_{16})$ because of join event e_{15}, and $\ell \in \mathsf{Before}_B(e_{16})$ because of acquire event e_{12}. In trace σ_2, $\ell \in \mathsf{Before}_U(e_3)$ because of acquire event e_2.

Now let us consider the computation of cross-races for the chunks in Figure 2. For $M \in \{D, E, F, G\}$, it is easy to see that $\mathsf{Race?}(M) = \mathsf{false}$, because each of these chunks only contain events of one thread. Let us look at the interesting pairs of events we considered in Example 2.3. The absence of race between e_1 and e_3 can be seen because $\mathsf{ALWr}_E(x) = \{1,2\}$ and $\mathsf{BFRd}_F(2,x) = \{2\}$, both of which have the thread 2 in common, and thus the intersection $\mathsf{ALWr}_E(x) \cap \mathsf{BFRd}_F(x)$ is non-empty. In fact, what this reasoning demonstrates is that there is no race between a $\mathtt{w}(x)$-event in E and a $\mathtt{r}(x)$-event in F. Similarly, the absence of a race between e_{13} and e_{16} can be seen because $\mathsf{ALWr}_F(y) \cap \mathsf{BFRd}_G(1,y) = \{2,\ell\} \cap \{1,2\} = \{2\} \neq \emptyset$. To reason about the events e_5 and e_{10}, observe that $\mathsf{ALWr}_F(y) = \{2,\ell\}$ and $\mathsf{BFWr}_D(y) = \{1,\ell\}$, both of which have the ℓ in common. Thus, we can conclude there is no race between a pair of $\mathtt{w}(y)$-events crossing the chunk FD.

Our reasoning also reveals the existence of HB-concurrent events. For example, $\mathsf{ALRd}_F(2,x) = \{2,\ell\}$, and $\mathsf{BFRd}_D(1,x) = \{1\}$. Since these sets are disjoint, it reveals that there are a pair of $\mathtt{r}(x)$-events (namely, e_3 and e_7) that are HB-concurrent; it is not a HB-race because these events are not conflicting (none of e_3 and e_7 is a write event). The race between e_{10} and e_{13} can be seen as follows. $\mathsf{ALWr}_A(y) = \{1\}$, and $\mathsf{BFWr}_B(y) = \{2,\ell\}$. We can see that there is a cross race in chunk AB, because these two sets are disjoint.

3.2 Computing Before and After sets

Our discussion in Section 3.1 suggests that if we manage to inductively compute the sets ALRd, ALWr, BFRd, and BFWr (Equation (1)) for each chunk in the grammar, then can determine if a chunk has a race using Equation (2). In this section we present such an inductive computation for these sets. We will only describe the computation of sets ALRd and BFRd. The computation of the sets ALWr and BFWr is similar and is presented completely in the Appendix.

The base case for non-terminals with rule $A \to a$, where a is an event, is straightforward. To conserve space, this definition is skipped here, but presented in the Appendix. So we focus on the inductive step when we have a non-terminal with rule $A \to BC$.

First consider the case of $\mathsf{ALRd}_A(t,x)$, which is equal to the after set $\mathsf{After}_A(e)$, where e is the last event amongst the read events $\mathsf{REvents}_A(t,x)$. If there are no read events in A by thread t on variable x (i.e., $\mathsf{REvents}_A(t,x) = \emptyset$), we will have $\mathsf{ALRd}_A(t,x) = \emptyset$. Otherwise, depending upon where the last read event e occurs in the chunk $[\![A]\!]$, we have two cases to consider. First, if this last read event e belongs to the chunk $[\![C]\!]$, then indeed $\mathsf{ALRd}_A(t,x) = \mathsf{After}_A(e)$. Observe that since $\{e' \mid e \leq^C_{\mathsf{HB}} e'\} = \{e' \mid e \leq^A_{\mathsf{HB}} e'\}$, we have $\mathsf{After}_A(e) = \mathsf{After}_C(e)$. Thus, in this case, $\mathsf{ALRd}_A(t,x) = \mathsf{ALRd}_C(t,x)$. The interesting case is when $\mathsf{REvents}_C(t,x)$ is empty and $\mathsf{REvents}_B(t,x) \neq \emptyset$, i.e., the last read event e belongs to the chunk B. Since $\{e' \in [\![B]\!] \mid e \leq^B_{\mathsf{HB}} e'\} \subseteq \{e' \in [\![A]\!] \mid e \leq^A_{\mathsf{HB}} e'\}$, we have $\mathsf{After}_B(e) \subseteq \mathsf{After}_A(e)$. Consider $e' \in [\![C]\!]$ such that $e \leq^A_{\mathsf{HB}} e'$. As in the discussion on cross-races in Section 3.1, this means there is a pair of events $e_1 \in [\![B]\!]$ and $e_2 \in [\![C]\!]$ such that $e \leq^B_{\mathsf{HB}} e_1$, $e_2 \leq^C_{\mathsf{HB}} e'$, and either (1) e_1, e_2 are events of the same thread, or (2) e_1 is a fork event and e_2 is an event of the child thread, or (3) e_1 is an event of a child thread and e_2 is a join event, or (4) e_1 is a release event and e_2 is an acquire event on the same lock. In each of these cases, e_1 witnesses the membership of some thread/lock u in $\mathsf{After}_B(e)$, and e' is HB-after the "first" event (namely e_2) of u in chunk C. The definition of what it means for an event to be "after"

the "first" event of a thread/lock u is subtle, and its definition is key in accurately capturing the intuitions just outlined.

For a non-terminal D and thread t, define

$$\mathsf{AF}_D(t) = \mathsf{After}_D(\mathsf{First}_D(\mathsf{ThEvents}^{\mathsf{join}}_D(t))) \tag{3}$$

where $\mathsf{ThEvents}^{\mathsf{join}}_D(t)$ is the set $\{e \in D \mid e = \langle t : o\rangle \text{ or } e = \langle t' : \mathtt{join}(t)\rangle\}$.

Similarly, for a lock ℓ, define

$$\mathsf{AF}_D(\ell) = \mathsf{After}_D(\mathsf{First}_D(\mathsf{AcqEvents}_D(\ell))) \tag{4}$$

where $\mathsf{AcqEvents}_D(\ell) = \{e \in D \mid e = \langle t : \mathsf{acq}(\ell)\rangle\}$. As before, we set $\mathsf{After}_D(\mathsf{undefined}) = \emptyset$.

We now formalize our intuitions in the following.

Lemma 3.5. *Let A be a non-terminal with rule $A \to BC$ and let $e \in [\![B]\!]$. Then*

$$\mathsf{After}_A(e) = \mathsf{After}_B(e) \cup \bigcup_{u \in \mathsf{After}_B(e)} \mathsf{AF}_C(u)$$

The proof of Lemma 3.5 is postponed to the Appendix. Its statement gives us the following inductive definition for $\mathsf{ALRd}_A(t,x)$.

$$\mathsf{ALRd}_A(t,x) = \begin{cases} \mathsf{ALRd}_C(t,x) & \text{if } \mathsf{ALRd}_C(t,x) \neq \emptyset \\ \mathsf{ALRd}_B(t,x) \cup \bigcup_{u \in \mathsf{ALRd}_B(t,x)} \mathsf{AF}_C(u) & \text{otherwise} \end{cases} \tag{5}$$

Notice that the second expression is $\emptyset$ if $\mathsf{ALRd}_B(t,x) = \emptyset$.

To complete the formal defintion of $\mathsf{ALRd}_A(x)$, we need to give an inductive definition for the sets AF. Again defining AF_A for $A \to a$ is straightforward from Equation (3) and Equation (4) and is defered to the Appendix. Consider the inductive step, of a non-terminal A with rule $A \to BC$ and let u be some thread/lock. If $\mathsf{First}_A(\mathsf{ThEvents}^{\mathsf{join}}_A(t)) \in [\![B]\!]$ then Lemma 3.5 forms the basis of our definition. However, it is possible that the set $\mathsf{ThEvents}^{\mathsf{join}}_B(t)$ is empty, while $\mathsf{First}_A(\mathsf{ThEvents}^{\mathsf{join}}_A(t)) \in [\![C]\!]$. In this case, $\mathsf{AF}_A(u) = \mathsf{AF}_C(u)$. A similar reasoning applies for a lock ℓ as well. Putting all these observations together, we get

$$\mathsf{AF}_A(u) = \mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u')$$

We conclude the description of the HB-algorithm for compressed traces, by outlining the inductive definition of the set $\mathsf{BFRd}_A(t,x)$ for thread t and variable x. As before, the first event $e \in [\![A]\!]$ of the kind $\langle t : \mathtt{r}(x)\rangle$ can either belong to $[\![B]\!]$ or to $[\![C]\!]$. When $e \in \mathsf{REvents}_B(t,x)$, we have $\mathsf{Before}_A(e) = \mathsf{Before}_B(e)$, since $\{e' | e' \leq^B_{\mathsf{HB}} e\} = \{e' \mid e' \leq^A_{\mathsf{HB}} e\}$. On the other hand, if $e \in [\![C]\!]$, in a manner similar to the case for $\mathsf{After}_B(e)$, we need to "compose" $\mathsf{Before}_C(e)$ with the before sets associated with the *last* events of threads/locks in chunk B. To carry this out, we need to define the set of events "before" the last event of a lock/thread in a chunk.

For a non-terminal D and a thread t,

$$\mathsf{BL}_D(t) = \mathsf{Before}_D(\mathsf{Last}_D(\mathsf{ThEvents}^{\mathsf{fork}}_D(t))) \tag{6}$$

where $\mathsf{ThEvents}^{\mathsf{fork}}_D(t) = \{e \in D \mid e = \langle t : o\rangle \text{ or } e = \langle t' : \mathtt{fork}(t)\rangle\}$.

For a lock ℓ,

$$\mathsf{BL}_D(\ell) = \mathsf{Before}_D(\mathsf{Last}_D(\mathsf{RelEvents}_D(\ell))) \tag{7}$$

where $\mathsf{RelEvents}_D(\ell) = \{e \in D \mid e = \langle t : \mathtt{rel}(\ell)\rangle\}$. We will assume $\mathsf{Before}_D(\mathsf{undefined}) = \emptyset$.

As in Lemma 3.5, the sets $\mathsf{BL}_B(u)$ can used to compute $\mathsf{Before}_A(e)$ for any event $e \in [\![C]\!]$.

Lemma 3.6. *Let A be a non-terminal with rule $A \to BC$ and let $e \in [\![C]\!]$. Then*

$$\mathsf{Before}_A(e) = \mathsf{Before}_C(e) \cup \bigcup_{u \in \mathsf{Before}_C(e)} \mathsf{BL}_B(u)$$

Using Lemma 3.6, the inductive definition of $\mathsf{BFRd}_A(x)$ is as follows.

$$\mathsf{BFRd}_A(t,x) = \begin{cases} \mathsf{BFRd}_B(t,x) & \text{if } \mathsf{BFRd}_B(t,x) \neq \emptyset \\ \mathsf{BFRd}_C(t,x) \cup \bigcup_{u \in \mathsf{BFRd}_C(t,x)} \mathsf{BL}_B(u) & \text{otherwise} \end{cases} \tag{8}$$

To complete the algorithm, we need to give the inductive definition of $\mathsf{BL}_A(u)$ for thread/lock u. Again the interesting case is the inductive case of a non-terminal A with rule $A \to BC$. As in the case of AF-sets, Lemma 3.6 needs to be adapted to account for the two possibilities: when the last event of a thread/lock u is in $[\![B]\!]$ or $[\![C]\!]$ to give us the following definition.

$$\mathsf{BL}_A(u) = \mathsf{BL}_C(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{BL}_C(u)} \mathsf{BL}_B(u')$$

This completes the description of the HB-algorithm on compressed traces. Its correctness is proved in the Appendix. For a trace σ compressed as an SLP of size g, this algorithm runs in time $O(g(T+L)^2(L+TV))$ and uses space $O(g(T+L)(L+TV))$, where T, L and V denote the number of threads, locks and variables in σ.

Example 3.7. We conclude this section by showing that the before and after sets given in Example 3.4 are computed correctly using our inductive characterization. We will focus on trace σ_1 and its SLP grammar in Figure 2. Let us consider the computation of $\mathsf{ALWr}_C(x)$. Observe that the last $\mathtt{w}(x)$-event in C is e_1. Further,

$$\mathsf{ALWr}_E(x) = \{1,2\} \quad \mathsf{AF}_F(1) = \emptyset \quad \mathsf{AF}_F(2) = \{2,\ell\}$$

Here $\mathsf{AF}_F(1) = \emptyset$ because there is no event of thread 1 in F. Using the inductive definition similar to Equation (5), we get $\mathsf{ALWr}_C(x) = \{1,2,\ell\}$ which is correct.

Next, consider the computation $\mathsf{BFWr}_B(y)$. Notice that the first $\mathtt{w}(y)$-event in B is e_{13}, which is in the chunk F. This immediately gives $\mathsf{BFWr}_B(y) = \mathsf{BFWr}_F(y) = \{2,\ell\}$ using a characterization similar to Equation (8).

4 LOCKSET ALGORITHM FOR COMPRESSED TRACES

Similar to our algorithm for detecting HB-races on compressed traces, we will formulate an algorithm for detecting violations of the lockset discipline on SLPs in an inductive fashion. The challenge here again is similar — violations occuring inside a chunk are also violations of any other chunk that contains this one, and detecting "cross" violations is key to checking all violations. In this section, we will outline these ideas in detail.

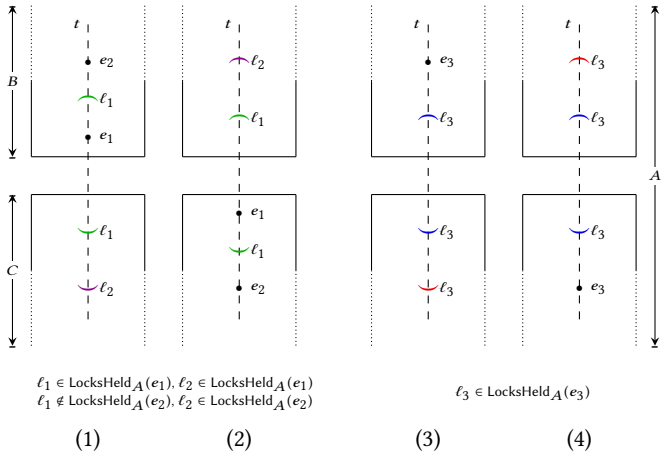

Figure 5: Unmatched acquire and release events protect all the events of the same thread in the neighboring chunk when not matched in the entire chunk ((1) and (2)). Re-entrant locks protect the neighboring chunk when the outermost unmatched acquire/release is unmatched ((3) and (4)). '$\frown\ \ell_i$' represents $\texttt{acq}(\ell_i)$, '$\smile\ \ell_i$' is $\texttt{rel}(\ell_i)$.

4.1 Cross violations

Recall that, for a thread t and variable x, $\text{LockSet}_\sigma(t, x)$ is the set of all the locks (including the dummy locks in $\mathcal{D}$) that *protect* every access event of x performed by t, in σ.

In this section, we will show how to compute $\text{LockSet}_A(t, x)$ for every non-terminal A and for every pair (t, x) of thread and variable, by inducting on the non-terminals in decreasing order of their rank. Checking if $\cap_{t \in \text{Threads}_A} \text{LockSet}_A(t, x) = \emptyset$ then follows easily.

The base case for non-terminals with rule $A \to a$ is straightforward, and is presented in the Appendix. Now consider the inductive step for non-terminals having rules of the form $A \to BC$. To understand what $\text{LockSet}_A(t, x)$ will be, it is useful to examine what $\text{LocksHeld}_A(e)$ for an event e looks like. Consider a data access event $e \in [\![B]\!]$ performed by thread t. Clearly, $\text{LocksHeld}_B(e) \subseteq \text{LocksHeld}_A(e)$. But are the equal? The answer turns out to be no. Suppose a lock ℓ which is released in $[\![C]\!]$ by thread t but does not have a *matching* acquire in $[\![A]\!]$ (and hence, neither in $[\![B]\!]$). Such a lock ℓ will protect all the events performed before it in $[\![A]\!]$. Thus trivially, it will enclose all the events performed by t in chunk $[\![B]\!]$. As a consequence, ℓ must be included in the set $\text{LocksHeld}_A(e)$ for every event $e \in B\restriction_t$. Lock ℓ_2 in Figure 5(1) illustrates this. Similarly, for an event $e \in [\![C]\!]$ performed by thread t, the set $\text{LocksHeld}_A(e)$ must additionally include locks which have been acquired by thread t in $[\![B]\!]$ but have not been matched in $[\![A]\!]$ (see lock ℓ_2 in Figure 5(2)). However, one must be careful. A lock ℓ which was released by t in C (at event $e_{\texttt{rel}(\ell)}$) and whose matching acquire is in B (event $e_{\texttt{acq}(\ell)}$), does not affect the locks held by any event in B — for those events $e \in B\restriction_t$ which were after $e_{\texttt{acq}(\ell)}$, ℓ was already in $\text{LocksHeld}_B(e)$, while for the events e before $e_{\texttt{acq}(\ell)}$, ℓ does not anyway protect e, and thus $\ell \notin \text{LocksHeld}_B(e)$. This is illustrated through lock ℓ_1 in Figure 5 [(1) and (2)].

In the presence of re-entrant locks, we need to account for another fact. Since locks can be acquired and released multiple times, a lock that is released more times in C (by thread t) than it is acquired in B (by thread t) will protect all events of t in B, because the outermost release is still unmatched in A. The same holds for locks that have been acquired more times than they are released in C. Both these scenarios are shown in Figure 5 [(3) and (4)].

To formalize the above notions, we will now introduce some notation. For a non-terminal D, let us first define the number of *unmatched* acquire events of lock ℓ in thread t as

$$\text{OpenAcq}_D(t, \ell) = |\{e = \langle t : \texttt{acq}(\ell)\rangle \in D \mid \texttt{match}(e) \notin [\![D]\!]\}| \quad (9)$$

and the number of release events as

$$\text{OpenRel}_D(t, \ell) = |\{e = \langle t : \texttt{rel}(\ell)\rangle \in D \mid \texttt{match}(e) \notin [\![D]\!]\}| \quad (10)$$

Our intuitions, as discussed above, can then be captured for the more complex case of re-entrant locks as follows.

Lemma 4.1. *Let A be a non-terminal with rule $A \to BC$. Let $e \in B\restriction_t$ and $e' \in C\restriction_{t'}$ be read/write events performed by threads t, t'. Then,*

$$\begin{aligned}\text{LocksHeld}_A(e) = \ & \text{LocksHeld}_B(e) \\ & \cup\{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\} \\ \text{LocksHeld}_A(e') = \ & \text{LocksHeld}_C(e') \\ & \cup\{\ell \mid \text{OpenAcq}_B(t', \ell) > \text{OpenRel}_C(t', \ell)\}\end{aligned}$$

Building on Lemma 4.1, we can now state the inductive definition of LockSet in terms of OpenAcq and OpenRel.

$$\begin{aligned}&\text{LockSet}_A(t, x) \\ &= \Big(\text{LockSet}_B(t, x) \cup \{\ell \mid \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell)\}\Big) \\ &\cap \Big(\text{LockSet}_C(t, x) \cup \{\ell \mid \text{OpenAcq}_B(t', \ell) > \text{OpenRel}_C(t', \ell)\}\Big)\end{aligned} \quad (11)$$

The base case for computing $\text{LockSet}_A(t, x)$ is trivial and is presented the Appendix for completeness.

Example 4.2. Consider the SLP for σ_2 from Figure 3. OpenAcq and OpenRel for the different non-terminals is given below.

$$\begin{aligned}&\text{OpenAcq}_W(1, \ell) = 1 \quad \text{OpenRel}_X(1, \ell) = 1 \\ &\text{OpenAcq}_X(2, \ell) = 1 \quad \text{OpenRel}_Z(2, \ell) = 1 \\ &\text{OpenAcq}_U(2, \ell) = 1 \quad \text{OpenRel}_V(2, \ell) = 1.\end{aligned}$$

The values for all other combinations are 0. Note how the unmatched acquire in W and the unmatched release in X on thread 1 get matched in the concatenated chunk U, giving $\text{OpenAcq}_U(1, \ell) = \text{OpenRel}_U(1, \ell) = 0$. This is essentially the insight we will explore to derive an inductive formulation of OpenAcq and OpenRel next, in Section 4.2.

Let us now see how LockSet computation takes place. First, $\text{LockSet}_W(1, x) = \{\Lambda, \Lambda_1\}$ since the only event of x in W is a read by thread 1. Also, $\text{LockSet}_X(1, y) = \{\ell, \Lambda_1\}$ as e_3 is protected by the (unmatched) release e_4 in X. In chunk Y, $\text{LockSet}_Y(2, x) = \{\Lambda, \Lambda_2\}$ and $\text{LockSet}_Y(2, x) = \{\Lambda_2\}$; interestingly, the locksets for Y does not reveal that both e_6 and e_7 are enclosed within the critical section of lock ℓ. On the other hand, using the inductive formulation discussed above, we can infer that $\text{LockSet}_V(2, y) = (\text{LockSet}_Y(2, y) \cup \{\ell\}) \cap \top$ which evaluates to $\{\Lambda_2, \ell\}$ as expected. The universal set $\top$ is described in Definition E.1 (See the Appendix). Again, the ℓ does not appear in $\text{LockSet}_U(1, x)$ eventhough it is unmatched in X,

because it gets matched with e_2 in W. This also follows from the inductive definition of $\text{LockSet}_U(1, x)$ because $\text{OpenRel}_X(1, \ell) \not> \text{OpenAcq}_W(1, \ell)$.

Given Equation (11), our inductive formulation will be complete once we can inductively compute the functions OpenAcq and OpenRel. We describe this next.

4.2 Computing OpenAcq and OpenRel

The aim of this section is to give inductive definitions of the functions OpenAcq and OpenRel. The base case for non-terminals having rules of the form $A \to a$ is straightforward and can be found in the Appendix for completeness.

In the inductive case we have a non-terminal A with production rule of the form $A \to BC$. For this case, let us first attempt to characterize the acquire events in $[\![A]\!]$ that have not been matched. Notice that if a lock is acquired (without a matching release) in the chunk $[\![C]\!]$, it would remain unmatched in the bigger chunk $[\![A]\!]$. In addition, the unmatched acquire events acquired in $[\![B]\!]$ whose matching release is not present in $[\![C]\!]$ will also contribute to the unmatched acquire events in $[\![A]\!]$. This reasoning is formally stated below

$$\begin{aligned}\text{OpenAcq}_A(t, \ell) = \ & \text{OpenAcq}_C(t, \ell) \\ & + \max\{0, \text{OpenAcq}_B(t, \ell) - \text{OpenRel}_C(t, \ell)\}\end{aligned} \tag{12}$$

Notice the use of the max operator in Equation (12). If the quantity $\text{OpenAcq}_B(t, \ell) - \text{OpenRel}_C(t, \ell)$ is negative, then there are more unmatched release events of ℓ in $C\restriction_t$, and they should be counted towards the unmatched release events for $[\![A]\!]$, instead of affecting the contribution of $[\![C]\!]$ towards the unmatched acquire events of $A\restriction_t$.

Using the same reasoning, the inductive formulation for OpenRel is stated below

$$\begin{aligned}\text{OpenRel}_A(t, \ell) = \ & \text{OpenRel}_B(t, \ell) \\ & + \max\{0, \text{OpenRel}_C(t, \ell) - \text{OpenAcq}_B(t, \ell)\}\end{aligned} \tag{13}$$

This completes the description of our algorithm for computing locksets and checking violations of lockset discipline for traces compressed using straight line programs. For a trace σ compressed as an SLP of size g, this algorithm runs in time $O(gTL(\log r + V))$ and uses space $O(gTL(\log r + V))$, where T, L and V are the number of threads, locks and variables respectively in σ, and r denotes the maximum number of times a thread acquires a lock without releasing it in σ.

5 EVALUATION

In order to gauge the effect of compression on the size of traces, and the subsequent effect on time taken to analyze these compressed traces for races, we conducted experiments on a large variety of benchmarks and evaluated our algorithms empirically. In this section, we describe the details of our implementation and experimental setup, and analyze the results of these experiments.

5.1 Implementation

Our algorithms for detecting races on compressed traces, discussed in Section 3 and Section 4 have been implemented in our tool ZipTrack. ZipTrack is written primarily in Java. Our techniques and algorithms are language independent and can be implemented to analyze executions of programs written in any language that uses threads, locks and shared memory for concurrent computation, such as C, C++ and Java. In our prototype implementation, we analyze traces generated by Java programs. ZipTrack firsts collects trace logs as sequence of events, which include read/write to memory locations, acquire/release of locks, and join/fork of threads. For this, we use the logging library provided by the commercial tool RVPredict [1]. After having generated the trace logs, ZipTrack calls the Sequitur algorithm (available at [2]) to compress these traces as straight line programs (see Section 2). ZipTrack then analyzes the generated SLPs to detect the presence of HB races and violations of the lockset discipline.

Optimizations. The SLPs generated using the Sequitur algorithm are not strictly CNF grammars; production rules in the grammar can have length > 2 as well. This is similar to the grammar shown in Figure 2, where both the non-terminals F and D have production rules of length 4. For detecting an HB-race on SLPs, ZipTrack employs the following optimizations that rely on existence of such long production rules. For a rule of the form $A \to a_1 a_2 \cdots a_k$, where each of $a_1, \ldots a_k$ are terminals, our tool ZipTrack uses a slight modification of the basic HB vector clock algorithm and uses the vector clock values to (i) determine if Race?(A) holds, and (ii) compute the various sets associated with A (such as ALRd_A, BFRd_A, etc.). For production rules where the right hand side has large contiguous sequences of terminals (or substrings), we introduce new production rules in the grammar, with fresh non-terminals corresponding to these long sequences. For example, for a rule such as $A \to b_1 \cdots b_k C d_1 \cdots d_m$, where b_is and d_is are terminals, we will introduce two non-terminals B and D, with production rules $B \to b_1 \cdots b_k$ and $D \to d_1 \cdots d_k$, and replace the production rule of A by $A \to BCD$. The idea here is to explicitly identify terminal-only rules to exploit the vector-clock optimization on more non-terminals.

5.2 Experimental Setup

Our experiments were conducted on an 8-core 2.6GHz 64-bit Intel Xeon(R) Linux machine, with HotSpot 1.8.0 64-But Server as the JVM and 30GB heap space.

To compare against Happens-Before and LockSet based analysis on uncompressed traces, we also implemented the standard Djit+ [40] vector clock algorithm and the FastTrack [18] algorithm that uses epoch optimizations to speedup vector-clock algorithm, and the Goldilocks [10] algorithm for HB race detection, as well as Eraser's lockset algorithm [46], as described in [40].

Our evaluation benchmarks (Column 1 in Table 1) are carefully chosen with the goal of being comprehensive, and have been primarily derived from [22]. The first set of small-sized (LOC ~ 50-300) benchmarks (account to pingpong) and are derived from IBM Contest benchmark suite [13]. The second set of medium sized (LOC ~ 3K) benchmarks (moldyn to raytracer) are derived from the Java Grande Forum benchmark suite [49]. The third set (derby to xalan) of benchmarks (LOC ~ 30K-500K) come from the DaCaPo benchmark suite (version 9.12) [6] and large real world software including Apache FTPServer, W3C Jigsaw web server, Apache Derby.

1	2	3	4
Program	Events	Grammar Size	Compr. Ratio
account	130	107	1.21
airline	137	132	1.04
array	47	47	1.0
boundedbuffer	337	194	1.74
bubblesort	4.2K	3.3K	1.29
bufwriter	11.8M	293	40238
critical	55	55	1.00
mergesort	3028	2795	1.08
pingpong	146	135	1.08
moldyn	164K	88K	1.86
montecarlo	7.2M	6.1M	1.18
raytracer	16.2K	14.6K	1.11
derby	1.3M	735K	1.83
eclipse	90.6M	42.5M	2.13
ftpserver	49K	30K	2.13
jigsaw	3M	908K	3.37
lusearch	216M	66.6K	3.25
xalan	122M	71M	1.70

Table 1: Benchmarks : Trace size, compressed grammar size and compression ratios.

1	2	3	4	5	6	7
	HB (ms)				LockSet (ms)	
Program	DJIT+	F.TRACK	Goldi.	Compr.	Eraser	Compr.
account	6	5	4	4	3	3
airline	8	4	5	6	2	1
array	5	4	4	3	3	1
boundedbuffer	8	8	2	18	3	2
bubblesort	14	13	12	92	2	4
bufwriter	20043	15538	36348	6	1	4
critical	3	4	3	5	2	1
mergesort	5	7	5	13	3	6
pingpong	10	9	9	3	2	1
moldyn	53	60	57	6	2	2
montecarlo	317	271	302	87	300	1
raytracer	58	32	32	315	25	133
derby	1006	1011	26015	592	848	<1
eclipse	34585	31527	3775764	17429	21737	1
ftpserver	49	44	91	23	34	1
jigsaw	2432	2309	1888	195	12	4
lusearch	1392	968	700	7	814	2
xalan	5183	3008	3709	109	2779	1

Table 2: Performance evaluation : Running times for different race detection techniques.

5.3 Results

Compression Ratio. To analyze the effect of compression on the size of traces, consider the compression ratios (ratio of the size of the original trace and the size of the grammar representation) shown in Column 4 in Table 1. The compression ratios are not significant for the small and medium sized benchmarks, barring boundedbuffer (compression ratio = 1.74), moldyn (compression ratio = 1.86) and the most notable bufwriter (compression ratio > 40,000). The compression ratios for the large benchmarks are impressive; as large as 3.25. This can be attributed to the fact that in large executions, the large amount of redundancies make them amenable to larger compression. Despite smaller lines of code in the source code of bufwriter, the size of the execution observed is quite large, and thus the excellent compression ratio.

HB race detection. Columns 2, 3 and 4 in Table 2 represent the time taken to detect the presence of an HB race by respectively, DJIT+, FASTTRACK and Goldilocks, while Column 5 denotes the time taken by our HB race detection algorithm for analyzing the traces compressed as SLPs. The FASTTRACK algorithm almost always performs better than both DJIT+ and Goldilocks. We will henceforth compare the speedup/slowdown of our algorithm (Column 5) over FASTTRACK (Column 3).

First, in the smaller examples (account - pingpong), the speed-up is not significant for most examples. This can be attributed to the low compression ratios, and significant initial set-up times. In particular, the bubblesort example has a significant slow-down. One noteworthy small example that shows the power of compression is bufwriter where the compression ratio and the resulting speedup for race detection is very high (> 2500x).

For the medium sized examples, the compression ratios range in $1.1 - 1.86$. The speedup for moldyn and montecarlo is about $3 - 4$x, while for raytracer, we encounter a large slowdown. A possible explanation for the degraded performance in both bubblesort and raytracer is that, while the first race pair (e_1, e_2) occurs much earlier in the uncompressed trace, the SLP generated is such that, in order to discover any race, the entire grammar needs to be processed.

The performance improvements for the large benchmarks are noteworthy and the speed ups shoot to the order of 140x. The FastTrack vector clock algorithm [18] is the gold standard for detecting HB races, and our evaluation indicates that analysis on compressed traces beats the advantages offered by vector-clocks and further epoch-like optimizations. In fact our algorithm is, in spirit, closer to the Goldilocks algorithm, for which the performance degradation deeply intensifies on larger benchmarks (also noted before in [18]). The speedups (over FASTTRACK) achieved by our approach, despite this similarity, must be attributed to the non-trivial compression ratios achieved. Overall, the average speed-up is about 2.9x over FASTTRACK, and around 200x over the Goldilocks algorithm.

Lockset violation detection. Columns 6 and 7 denote the time for detecting lockset violations on uncompressed and compressed traces respectively. Since, the compression on smaller examples is not large, we can observe that the speedup in such examples is not extraordinary. However, there is little or almost no slowdown. For the medium and large examples, ZIPTRACK detects violations of lockset discipline on compressed traces much faster than on uncompressed traces. In fact, the speed-ups shoot upto more than $20,000$x, and the time taken is almost always of the order of a few milliseconds. The average speed-up achieved over the Eraser algorithm is around 173x.

Clearly, these large real-world examples illustrate the benefit of compression; not only do they yield smaller storage spaces, but they also result in a more efficient analysis.

6 CONCLUSIONS

We considered the problem of detecting races in traces compressed by SLPs. We presented algorithms that detect HB-races and violations of the lockset discipline in time that is linear in the size of the compressed traces. Experimental evaluation of our implementation of these algorithms in the tool ZIPTRACK, demonstrated that analyzing compressed traces can lead to significant speedups.

REFERENCES

[1] 2017. RV-Predict, Runtime Verification. https://runtimeverification.com/predict/. (2017). Accessed: 2017-11-01.

[2] 2017. Sequitur: Inferring Hierarchies From Sequences. http://www.sequitur.info/. (2017). Accessed: 2017-08-01.

[3] M. Abadi, C. Flanagan, and S.N. Freund. 2006. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems* 28, 2 (2006), 207–255.

[4] J.L. Balcázar. 1996. The complexity of searching implicit graphs. *Artificial Intelligence* 86, 1 (1996), 171–188.

[5] Swarnendu Biswas, Jipeng Huang, Aritra Sengupta, and Michael D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 28–39.

[6] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frmpton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the ACM SIGPLA Conference on Object-Oriented Programming Systems, Languages, and Applications*. 169–190.

[7] C. Boyapati, R. Lee, and M. Rinard. 2002. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Lanaguages, and Applications*. 211–230.

[8] G.-I. Cheng, M. Feng, C.E. Leiserson, K.H. Randall, and A.F. Stark. 1998. Detecting Data Races in Cilk Programs That Use Locks. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. 298–309.

[9] B. Das, P. Scharpfenecker, and J. Torán. 2014. Succinct encodings of graph isomorphism. In *Proceedings of the Internation Conference on Languages and Automata Theory and Applications*. 285–296.

[10] T. Elmas, S. Qadeer, and S. Tasiran. 2007. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–255.

[11] D. Engler and K. Ashcraft. 2003. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 237–252.

[12] M. Eslamimehr and J. Palsberg. 2014. Sherlock: Scalable deadlock detection for concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 353–365.

[13] E. Farchi, Y. Nir, and S. Ur. 2003. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the International Symposium on Parallel and Distributed Processing*.

[14] J. Feigenbaum, S. Kannan, M.Y. Vardi, and M. Viswanathan. 1998. Complexity of Problems on Graphs Represented as OBDDs. In *Proceedings of the Annual Symposium on Theoretical Aspects of Computer Science*. 216–226.

[15] M. Feng and C.E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. 1–11.

[16] C.J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*. 56–66.

[17] C. Flanagan and S.N. Freund. 2000. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 219–232.

[18] C. Flanagan and S.N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–133.

[19] C. Flanagan and S.N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 1–8.

[20] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. 2008. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 293–303.

[21] H. Galperin and A. Wigderson. 1983. Succinct Representations of Graphs. *Information and Control* 56, 3 (1983), 183–198.

[22] J. Huang, P.O. Meredith, and G. Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 337–348.

[23] J. Huang and A.K. Rajagopalan. 2016. Precise and maximal race detection from incomplete traces. In *Proceedings of the ACM SIGPLAN International Conference on Object-oriented Programming, Systems, Languages, and Applications*. 462–476.

[24] S.F. Kaplan, Y. Smaragdakis, and P.R. Wilson. 2003. Flexible reference trace reduction for VM simulations. *ACM Transactions on Modeling and Computer Simulation* 13, 1 (2003), 1–38.

[25] J.C. Kieffer and E.-H. Yang. 2000. Grammar-based codes: a new class of universal lossless source codes. *IEEE Transactions on Information Theory* 46, 3 (2000), 737–754.

[26] J.C. Kieffer, E.-H. Yang, G.J. Nelson, and P. Cosman. 2000. Universal lossless compression via multilevel pattern matching. *IEEE Transactions on Information Theory* 46, 4 (2000), 1227–1245.

[27] D. Kini, U. Mathur, and M. Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 157–170.

[28] A. Kinneer, M.B. Dwyer, and G. Rothermel. 2007. Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*. 51–52.

[29] L. Lamport. 1978. Time, Clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[30] N.J. Larsson and A. Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[31] P. Liu, O. Tripp, and X. Zhang. 2016. IPA: Improving Predictive Analysis with Pointer Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*. 59–69.

[32] A. Lozano and J.L. Balcázar. 1986. The complexity of graph problems for succinctly represented graphs. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*. 277–286.

[33] F. Mattern. 1988. Virtual time and Global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. 215–226.

[34] A. Milenković and M. Milenković. 2007. An Efficient Single-Pass Trace Compression Technique Utilizing Instruction Streams. *ACM Transactions on Modeling and Computer Simulation* 17, 1 (2007).

[35] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*. 267–280.

[36] M. Naik, A. Aiken, and J. Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.

[37] C.G. Nevill-Manning. 1996. *Inferring Sequential Structure*. Ph.D. Dissertation. University of Waikato.

[38] C.G. Nevill-Manning and I.H. Witten. 1997. Identifying hierarchical structure in sequences: A linear time algorithm. *Journal of Artificial Intelligence* 7 (1997), 67–82.

[39] C.H. Papadimitriou and M. Yannakakis. 1986. A note on succinct representations of graphs. *Information and Control* 71, 3 (1986), 181–185.

[40] E. Pozniansky and A. Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 179–190.

[41] P. Pratikakis, J.S. Foster, and M. Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems* 33, 1 (2011), 3:1–3:55.

[42] C.v. Praun and T.R. Gross. 2001. Object race detection. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. 70–82.

[43] C. Radoi and D. Dig. 2013. Practical static race detection for Java parallel loops. In *Proceedings of the International Symposium on Software Testing and Analysis*. 178–190.

[44] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 531–542.

[45] M. Said, C. Wang, Z. Yang, and K. Sakallah. 2011. Generating Data Race Witnesses by an SMT-based Analysis. In *Proceedings of the International Conference on NASA Formal Methods*. 313–327.

[46] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. 1997. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 27–37.

[47] K. Sen. 2008. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 11–21.

[48] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 387–400.

[49] L.A. Smith, J.M. Bull, and J. Obdrzálek. 2001. A Parallel Java Grande benchmark suite. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. 8–8.

[50] R. Surendran and V. Sarkar. 2016. Dynamic determinacy race detection for task parallelism with futures. In *Proceedings of the International Conference on Runtime Verification*. 368–385.

[51] H. Veith. 1996. Succinct Representation, Leaf Languages, and Projection Reductions. In *Proceedings of the IEEE Conference on Computational Complexity*. 118–126.

[52] J.W. Voung, R. Jhala, and S. Lerner. 2007. RELAY: Static race detection on millions of lines of code. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 205–214.

[53] C. Wang, S. Kundu, M. Ganai, and A. Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *Proceedings of the World Congress on Formal Methods*. 256–272.

[54] T.A. Welch. 1984. A Technique for High-Performance Data Compression. *Computer* 17, 6 (1984), 8–19.

[55] E. Yahav. 2001. Verifying Safety Properties of Concurrent Java Programs Using 3-valued Logic. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 27–40.

[56] En-Hui Yang and J. C. Kieffer. 2000. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. I. Without context models. *IEEE Transactions on Information Theory* 46, 3 (2000), 755–777.

[57] A. Yoga, S. Nagarakatte, and A. Gupta. 2016. Parallel Data Race Detection for Task Parallel Programs with Locks. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 833–845.

[58] S. Zhan and J. Huang. 2016. ECHO: Instantaneous in situ race detection in the IDE. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 775–786.

[59] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

A COMPLEXITY ANALYSIS OF STANDARD ALGORITHMS

Complexity of Vector Clock Algorithm.

Theorem A.1. *Consider a trace σ of length n with T threads, L locks, and V variables. The vector clock algorithm for detecting* HB*-races runs in time $O(nT \log n)$ and uses space $O((V + L + T)T \log n)$.*

Proof Sketch. Each vector clock is a vector with T components, where each entry is a natural number $\leq n$. Thus, each vector clock uses space $O(T \log n)$ and each vector clock operation takes $O(T \log n)$ time. Processing each event in the trace involves constantly many vector clock operations. Finally, the algorithm maintains a vector clock for each thread, lock, and variable. All these observations together give us the bounds in the theorem. □

Complexity of Goldilocks Algorithm.

Theorem A.2. *Consider a trace σ length n with T threads, L locks, and V variables. The Goldilocks algorithm for detecting* HB*-races runs in time $O(n(L + TV))$ and uses space $O(TV(T + L))$.*

Proof Sketch. Let us compute the space requirement first. For every variable x the algorithm maintains the set $\mathsf{GLS}^{\mathsf{W}}(x)$, which has at most $T + L$ elements. Similarly, for every (t, x) it maintains the set $\mathsf{GLS}^{\mathsf{R}}(t, x)$ of size at most $T + L$. Thus, the total space usage is $O(V(T + L) + TV(T + L)) = O(TV(T + L))$.

Now let us compute the running time. When a write event $\langle t : \mathtt{w}(x)\rangle$ is performed, the algorithm checks if $t \in \mathsf{GLS}^{\mathsf{W}}(x)$, or if $\bigvee_{t' \in \mathsf{Threads}} t \in \mathsf{GLS}^{\mathsf{R}}(t', x)$. This check takes time $O(1 + T)$. After this check, the algorithm updates $\mathsf{GLS}^{\mathsf{W}}(x)$ to the singleton set $\{t\}$. This takes time $O(T + L)$ assuming we use a data structure like *bit-vectors* of length $T + L$ to maintain these sets. The total time, thus, for processing a write event is $O(T + L)$. When a read event $\langle t : \mathtt{r}(x)\rangle$ is performed, the algorithm check if $t \in \mathsf{GLS}^{\mathsf{W}}(x)$, and then updates $\mathsf{GLS}^{\mathsf{R}}(x)$ to the set $\{t\}$. Thus, a read event takes time $O(T + L)$.

For an acquire event $\langle t : \mathsf{acq}(\ell)\rangle$, the algorithm, (i) for every variable x for which $\ell \in \mathsf{GLS}^{\mathsf{W}}(x)$, adds t to $\mathsf{GLS}^{\mathsf{W}}(x)$, and (ii) for every pair (x, t') for which $\ell \in \mathsf{GLS}^{\mathsf{R}}(t', x)$, adds t to $\mathsf{GLS}^{\mathsf{R}}(t', x)$. This takes $O(V(1+1)+TV(1+1)) = O(TV)$ time. For a release event, the converse happens; if the thread t is present in any of these sets, the lock ℓ is added to them. Again, this is $O(TV)$ time. A similar update happens on a fork or a join, and the same bound applies for such events too. So the overall time complexity is $O(n(T+L+TV)) = O(n(L + TV))$. □

Complexity of Eraser's Lockset Algorithm.

Theorem A.3. *Let σ be a trace σ of length n with T threads, L locks, and V variables. Let r be the maximum number of times a thread acquires a lock without releasing it. The Eraser algorithm detects violations of the locket discipline on σ in time $O(n(L + \log r))$ and uses space $O(TL \log r + V(T + L))$.*

Proof Sketch. Let us analyze the space usage first. For every thread t and every lock ℓ, the algorithm maintains an integer value representing the number of unmatched acquires of lock ℓ in thread t. Since each of these values do not exceed r, this contributes $O(TL \log r)$ the space usage. Additionally, for every variable x, the algorithm maintains a set of locks, whose size can be at most $L + |\mathcal{D}| = L + T + 1$. This amounts to a space usage of $O(V(T + L + 1))$. The total space usage is thus $O(TL \log r + V(T + L + 1)) = O(TL \log r + V(T + L))$.

Let us now analyze the running time. For every event corresponding to an acquire/release of lock ℓ by thread t, the algorithm updates the number of unmatched acquires of ℓ in t. This takes time $O(\log r)$. For every read/write event $\langle t : a(x)\rangle$ $(a \in \{\mathtt{r}(), \mathtt{w}()\})$, the algorithm checks if (i) t is in the lockset of x, (ii) if the dummy lock Λ is contained in the lockset of x (when $a = \mathtt{r}()$), (iii) if the disjunction of two sets, of size at most L, is empty. This takes a time of $O(L)$. The total time, therefore, is $O(n(L + \log r))$. □

B PROOFS FROM SECTION 3

Proof of Lemma 3.1.

Lemma 3.1. *Consider events $e \in [\![B]\!]$ and $e' \in [\![C]\!]$. $e \leq^{BC}_{\mathsf{HB}} e'$ iff $\mathrm{After}_B(e) \cap \mathrm{Before}_C(e') \neq \emptyset$.*

Proof. Recall that if $e \leq^{BC}_{\mathsf{HB}} e'$ then there is a sequence $e = e_1, e_2, \ldots e_n = e'$ that satisfies the conditions in Definition 2.2. Let j be such that for all $i \leq j$, $e_i \in [\![B]\!]$ and for all $i \geq j + 1$, $e_i \in [\![C]\!]$. Also, we have $e \leq^{B}_{\mathsf{HB}} e_j$ and $e_{j+1} \leq^{C}_{\mathsf{HB}} e'$.

If e_j, e_{j+1} satisfy condition (1) of Definition 2.2, then $t \in \mathrm{After}_B(e) \cap \mathrm{Before}_C(e')$, where t is the thread performing both e_j and e_{j+1}. If e_j, e_{j+1} satisfy condition (2), then $\ell \in \mathrm{After}_B(e) \cap \mathrm{Before}_C(e')$, where ℓ is the lock released/acquired by event e_j and e_{j+1}, respectively. Finally, if e_j, e_{j+1} satisfy condition (3) or (4), $t' \in \mathrm{After}_B(e) \cap \mathrm{Before}_C(e')$, where t' is the thread forked by e_j and the one performing e_{j+1} or the child thread joined in e_{j+1}.

On the other hand if $\mathrm{After}_B(e) \cap \mathrm{Before}_C(e') \neq \emptyset$, then we can demonstrate that $e \leq^{BC}_{\mathsf{HB}} e'$. Let $u \in \mathrm{After}_B(e) \cap \mathrm{Before}_C(e')$. Let us first consider the case when u is a lock ℓ. Since $u = \ell \in \mathrm{After}_B(e)$, then there is an event $e_1 \in [\![B]\!]$ such that e_1 is $\mathtt{rel}(\ell)$ event and $e \leq^{B}_{\mathsf{HB}} e_1$. Similarly, since $u = \ell \in \mathrm{Before}_C(e')$, there is an event $e_2 \in [\![C]\!]$ such that e_2 is an $\mathsf{acq}(\ell)$ event and $e_2 \leq^{C}_{\mathsf{HB}} e'$. Putting these together, we get that $e \leq^{BC}_{\mathsf{HB}} e'$. Let us now consider the case when $u = t$. Again, $u = t \in \mathrm{After}_B(e)$ means that there is an event $e_1 \in [\![B]\!]$ such that $e \leq^{B}_{\mathsf{HB}} e_1$ and either e_1 is an event performed by t, or e_1 is a $\mathtt{fork}(t)$ event. Similary, $u = t \in \mathrm{Before}_C(e')$, means that there is an event $e_2 \in [\![C]\!]$ such that $e_2 \leq^{C}_{\mathsf{HB}} e'$ and either e_2 is an event performed by t, or e_2 is a $\mathtt{join}(t)$ event. In all these four cases, we can conclude that $e \leq^{BC}_{\mathsf{HB}} e'$. □

Proof of Lemma 3.2.

Let us first note the following observation:

Lemma B.1. *Let σ be a trace σ, and let e_1, e_2 be events such that $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$. Then, $\mathrm{After}_\sigma(e_2) \subseteq \mathrm{After}_\sigma(e_1)$ and $\mathrm{Before}_\sigma(e_1) \subseteq \mathrm{Before}_\sigma(e_2)$.*

Proof. The set $\mathrm{After}_\sigma(e_2)$ is

$$\begin{aligned}\mathrm{After}_\sigma(e_2) = \;& \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t : o\rangle.\ e_2 \leq^{\sigma}_{\mathsf{HB}} e'\}\\ \cup\; & \{t \in \mathsf{Threads}(\sigma) \mid \exists e' = \langle t' : \mathtt{fork}(t)\rangle.\ e_2 \leq^{\sigma}_{\mathsf{HB}} e'\}\\ \cup\; & \{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathtt{rel}(\ell)\rangle.\ e \leq^{\sigma}_{\mathsf{HB}} e'\}\end{aligned}$$

Since the HB relation $\leq^{\sigma}_{\mathsf{HB}}$ is transitive, we have that for all the events e' for which $e_2 \leq^{\sigma}_{\mathsf{HB}} e'$, we must have $e_1 \leq^{\sigma}_{\mathsf{HB}} e'$. Clearly, $\mathsf{After}_\sigma(e_2) \subseteq \mathsf{After}_\sigma(e_1)$.

A similar reasoning proves $\mathsf{Before}_\sigma(e_1) \subseteq \mathsf{Before}_\sigma(e_2)$. □

Let us now prove Lemma 3.2

Lemma 3.2. *If there is no HB race in $[\![B]\!]$ or in $[\![C]\!]$, and if there is an HB race between events $e \in [\![B]\!]$ and $e' \in [\![C]\!]$ then, there is an HB race between last^{e}_{B} and $\mathsf{first}^{e'}_{C}$, where*

$$\mathsf{last}^{e}_{B} = \begin{cases} \mathsf{Last}_B(\mathsf{REvents}_B(t,x)) & \text{if } e = \langle t : \mathtt{r}(x)\rangle \\ \mathsf{Last}_B(\mathsf{WEvents}_B(x)) & \text{if } e = \langle t : \mathtt{w}(x)\rangle \end{cases}$$

and

$$\mathsf{first}^{e'}_{C} = \begin{cases} \mathsf{First}_C(\mathsf{REvents}_C(t,x)) & \text{if } e' = \langle t' : \mathtt{r}(x)\rangle \\ \mathsf{First}_C(\mathsf{WEvents}_C(x)) & \text{if } e' = \langle t' : \mathtt{w}(x)\rangle \end{cases}$$

Proof. First, notice that, for a given trace σ, and for $(t,x) \in \mathsf{Rd}(\sigma)$, the elements of the set $\mathsf{REvents}_\sigma(t,x)$ are totally ordered by the relation $\leq^{\sigma}_{\mathsf{HB}}$. This follows from the rule (1) of Definition 2.2. Thus, for an event $e \in \mathsf{REvents}_\sigma(t,x)$, we have $\mathsf{First}_\sigma(\mathsf{REvents}_\sigma(t,x)) \leq^{\sigma}_{\mathsf{HB}} e \leq^{\sigma}_{\mathsf{HB}} \mathsf{Last}_\sigma(\mathsf{REvents}_\sigma(t,x))$. Using Lemma B.1, we must also have

- $\mathsf{After}_\sigma(\mathsf{Last}_\sigma(\mathsf{REvents}_\sigma(t,x))) \subseteq \mathsf{After}_\sigma(e)$,
- $\mathsf{After}_\sigma(e) \subseteq \mathsf{After}_\sigma(\mathsf{First}_\sigma(\mathsf{REvents}_\sigma(t,x)))$,
- $\mathsf{Before}_\sigma(\mathsf{First}_\sigma(\mathsf{REvents}_\sigma(t,x))) \subseteq \mathsf{Before}_\sigma(e)$, and
- $\mathsf{Before}_\sigma(e) \subseteq \mathsf{Before}_\sigma(\mathsf{Last}_\sigma(\mathsf{REvents}_\sigma(t,x)))$

Next observe that for a trace σ, if $\mathsf{Race?}(\sigma)$ is false, the elements of the set $\mathsf{WEvents}_\sigma(x)$ for some $x \in \mathsf{Wr}(\sigma)$ must be totally ordered by $\leq^{\sigma}_{\mathsf{HB}}$, and thus for such a trace σ,

- $\mathsf{After}_\sigma(\mathsf{Last}_\sigma(\mathsf{WEvents}_\sigma(x))) \subseteq \mathsf{After}_\sigma(e)$,
- $\mathsf{After}_\sigma(e) \subseteq \mathsf{After}_\sigma(\mathsf{First}_\sigma(\mathsf{WEvents}_\sigma(x)))$,
- $\mathsf{Before}_\sigma(\mathsf{First}_\sigma(\mathsf{REvents}_\sigma(x))) \subseteq \mathsf{Before}_\sigma(e)$, and
- $\mathsf{Before}_\sigma(e) \subseteq \mathsf{Before}_\sigma(\mathsf{Last}_\sigma(\mathsf{WEvents}_\sigma(x)))$

Now since there is a race between $e \in [\![B]\!]$ and $e' \in [\![C]\!]$, by Lemma 3.1, we have $\mathsf{After}_B(e) \cap \mathsf{Before}_C(e') = \emptyset$. Clearly, this means that $\mathsf{After}_B(\mathsf{last}^{e}_{B}) \cap \mathsf{Before}_C(\mathsf{first}^{e'}_{C}) = \emptyset$. Again, from Lemma 3.1, we have that $\mathsf{last}^{e}_{B} || _{\mathsf{HB}} \mathsf{first}^{e'}_{C}$. Also, $\mathsf{last}^{e}_{B} \asymp \mathsf{first}^{e'}_{C}$ because $e \asymp e'$. Thus, there is an HB race between last^{e}_{B} and $\mathsf{first}^{e'}_{C}$. □

Proof of Theorem 3.3.

Before we prove Theorem 3.3, let us first note the following simple observation.

Proposition B.2. *Let σ be a trace and $e_1, e_2 \in \sigma$. $e_1 \leq^{\sigma}_{\mathsf{HB}} e_2$ iff for any trace $\sigma' = \sigma_1\sigma\sigma_2$, we have $e_1 \leq^{\sigma'}_{\mathsf{HB}} e_2$.*

Informally, this means that the presence/absence of an HB-race between two events in a trace σ, does not get affected by the "context" in which σ is placed. The proof of the above proposition follows easily from Definition 2.2, and is skipped.

We now move on to the proof of Theorem 3.3.

Theorem 3.3. *For any non-terminal A, $\mathsf{Race?}(A) = \mathsf{true}$ if and only if there are events $e_1, e_2 \in [\![A]\!]$ such that $e_1 \asymp e_2$ and $e_1 || _{\mathsf{HB}} e_2$.*

Proof. The proof is by induction on the ranking of the non-terminals. In the base case, the theorem clearly holds. In the inductive step, consider a non-terminal $A \to BC$. Suppose $\mathsf{Race?}(A) = \mathsf{true}$. Based on Equation (2), this means either $\mathsf{Race?}(B)$ or $\mathsf{Race?}(C)$ are true or some pair of after and before sets are disjoint. If $\mathsf{Race?}(B)$ (or $\mathsf{Race?}(C)$) is true, then by induction hypothesis, there is an HB-race in $[\![B]\!]$ (or $[\![C]\!]$), and so there is a race in $[\![A]\!]$. This follows from Proposition B.2. If not (that is, if both $\mathsf{Race?}(B)$ and $\mathsf{Race?}(C)$ are false), the condition in one of line 2, 3 or 4 in Equation (2) is true. The proof is the same no matter which of these conditions hold. Therefore, let us assume without loss of generality, that $\mathsf{ALRd}_B(t,x) \cap \mathsf{BFWr}_C(x) = \emptyset$ for some thread t and variable x. Since $(t,x) \in \mathsf{REvents}_B$ and $x \in \mathsf{WEvents}_C$, both $\mathsf{ALRd}_B(t,x)$ and $\mathsf{BFWr}_C(x)$ are non-empty sets. Thus, both the events e, e' given by $e = \mathsf{Last}_B(\mathsf{REvents}_B(t,x))$ and $e' = \mathsf{First}_C(\mathsf{WEvents}_C(x))$ are well defined. Also, by definition, $\mathsf{After}_B(e) = \mathsf{ALRd}_B(t,x)$ and $\mathsf{Before}_C(e') = \mathsf{BFWr}_C(x)$. Thus, we have $\mathsf{After}_B(e) \cap \mathsf{Before}_C(e') = \emptyset$. Then, by Lemma 3.1, $e || _{\mathsf{HB}} e'$ in the chunk $[\![BC]\!] = [\![A]\!]$. Also, e, e' are clearly conflicting (read and write of same variable). Thus, (e, e') is an HB race in the chunk A.

Conversely, suppose there are a pair of events $e_1, e_2 \in [\![A]\!]$ such that $e_1 \asymp e_2$ and $e_1 || _{\mathsf{HB}} e_2$. Without loss of generality, let us assume that e_1 is $\mathtt{r}(x)$-event, and e_2 is a $\mathtt{w}(x)$-event. If $e_1, e_2 \in [\![B]\!]$ or $e_1, e_2 \in [\![C]\!]$ then by induction hypothesis, $\mathsf{Race?}(B)$ or $\mathsf{Race?}(C)$ is true. By Equation (2) this would imply that $\mathsf{Race?}(A)$ is true. Now consider the case when there are no races in B or C, and there are $e_1 \in [\![B]\!]$ and $e_2 \in [\![C]\!]$ which are in race. Without loss of generality, let us assume that $e_1 = \langle t : \mathtt{r}(x)\rangle$ is a read event, and $e_2 = \langle t' : \mathtt{w}(x)\rangle$ is a write event on the same variable x. Then, by Lemma 3.2, the events $e = \mathsf{last}^{e_1}_{B} = \mathsf{Last}_B(\mathsf{REvents}_B(t,x))$ and $e' = \mathsf{first}^{e_2}_{C} \mathsf{First}_C(\mathsf{WEvents}_C(x))$ are in race too. Then, again using Lemma 3.1, we must have that $\mathsf{After}_B(e) \cap \mathsf{Before}_C(e') = \emptyset$. Now, by definition of the sets ALRd and BFWr, we have $\mathsf{ALRd}_B(t,x) = \mathsf{After}_B(e)$ and $\mathsf{BFWr}_C(x) = \mathsf{Before}_C(e')$. Thus, clearly, $\mathsf{Race?}(A) = \mathsf{true}$. □

Proof of Lemma 3.5 and Lemma 3.6.

The proof of Lemma 3.5 and Lemma 3.6 depend on a simple observation that we establish first.

Lemma B.3. *Let A be a non-terminal with rule $A \to BC$. Let $e_1 \in [\![B]\!]$ and $e_2 \in [\![C]\!]$. The following three statements are equivalent.*

(1) $e_1 \leq^{A}_{\mathsf{HB}} e_2$.

(2) There is $u \in \mathsf{After}_B(e_1)$ such that first^{u}_{C} is defined and $\mathsf{first}^{u}_{C} \leq^{C}_{\mathsf{HB}} e_2$.

(3) There is $u \in \mathsf{Before}_C(e_2)$ such that last^{u}_{B} is defined and $e_1 \leq^{B}_{\mathsf{HB}} \mathsf{last}^{u}_{B}$.

where

$$\mathsf{first}^{u}_{C} = \begin{cases} \mathsf{First}_C(\mathsf{ThEvents}^{\mathsf{join}}_{C}(u)) & \text{if } u \in \mathsf{Threads}(C) \\ \mathsf{First}_C(\mathsf{AcqEvents}_C(u)) & \text{if } u \in \mathsf{Locks}(C) \end{cases}$$

and

$$\mathsf{last}^{u}_{B} = \begin{cases} \mathsf{Last}_B(\mathsf{ThEvents}^{\mathsf{fork}}_{B}(u)) & \text{if } u \in \mathsf{Threads}(B) \\ \mathsf{Last}_B(\mathsf{RelEvents}_B(u)) & \text{if } u \in \mathsf{Locks}(B) \end{cases}$$

Proof. First observe that set of events $\mathsf{ThEvents}^{\mathsf{join}}_C(t)$, for some $t \in \mathsf{Threads}(C)$, is totally ordered with respect to $\leq^C_{\mathsf{HB}}$. This is because all the events in this set are either of the form $\langle t : o\rangle$ or $\langle t' : \mathtt{join}(t)\rangle$, and (i) $\leq^C_{\mathsf{HB}}$ orders all events in a single thread thread (Definition 2.2 (1)), and (ii) also orders all events of a thread before the join event of that thread (Definition 2.2 (4)). Similarly, the events of the sets $\mathsf{ThEvents}^{\mathsf{fork}}_B(t)$, $\mathsf{AcqEvents}_C(\ell)$ and $\mathsf{RelEvents}_B(\ell)$ are also ordered totally with respect to $\leq^B_{\mathsf{HB}}$, $\leq^C_{\mathsf{HB}}$ and $\leq^B_{\mathsf{HB}}$ respectively.

Let us start by showing (1) $\Rightarrow$ (2). As in the discussion on cross-races in Section 3.1, this means there is a pair of events $e_3 \in [\![B]\!]$ and $e_4 \in [\![C]\!]$ such that $e_1 \leq^B_{\mathsf{HB}} e_3$, $e_4 \leq^C_{\mathsf{HB}} e_2$, and one of the following four conditions holds.

- e_3 and e_4 are events performed by the same thread (say) t. Thus, $\mathsf{ThEvents}^{\mathsf{join}}_C(t)$ is non-empty (contains e_4), and first^t_C is defined. Since $e_1 \leq^B_{\mathsf{HB}} e_3$, we have $t \in \mathsf{After}_B(e_1)$. Moreover, $\mathsf{first}^t_C \leq^C_{\mathsf{HB}} e_4 \leq^C_{\mathsf{HB}} e_2$, establishing the claim.
- $e_3 = \langle t' : \mathtt{fork}(t)\rangle$ and $e_4 = \langle t : o\rangle$ for some t', t, o. Thus, $\mathsf{ThEvents}^{\mathsf{join}}_C(t)$ is non-empty (contains e_4), and first^t_C is defined. Then by definition, we have $t \in \mathsf{After}_B(e_1)$, and $\mathsf{first}^t_C \leq^C_{\mathsf{HB}} e_4 \leq^C_{\mathsf{HB}} e_2$, which proves the claim.
- $e_3 = \langle t : o\rangle$ and $e_4 = \langle t' : \mathtt{join}(t)\rangle$, for some t, t', and o. Thus, $\mathsf{ThEvents}^{\mathsf{join}}_C(t)$ is non-empty (contains e_4), and first^t_C is defined. We have $t \in \mathsf{After}_B(e_1)$. And, $\mathsf{first}^t_C \leq^C_{\mathsf{HB}} e_4 \leq^C_{\mathsf{HB}} e_2$.
- $e_3 = \langle t : \mathtt{rel}(\ell)\rangle$ and $e_4 = \langle t : \mathtt{acq}(\ell)\rangle$ for some t, ℓ. Thus, $\mathsf{AcqEvents}_C(\ell)$ is non-empty (contains e_4), and first^ℓ_C is defined. In this case we have $\ell \in \mathsf{After}_B(e_1)$, and $\mathsf{first}^\ell_C \leq^C_{\mathsf{HB}} e_4 \leq^C_{\mathsf{HB}} e_2$, which establishes the claim as well.

Now we show that (2) $\Rightarrow$ (3). Let us assume that there is $u \in \mathsf{After}_B(e_1)$ such that first^u_C is defined and $\mathsf{first}^u_C \leq^C_{\mathsf{HB}} e_2$. Since $u \in \mathsf{After}_B(e_1)$, let e_3 be the event that "witnesses" the membership of $u \in \mathsf{After}_B(e_1)$. By definition, we have $e_1 \leq^B_{\mathsf{HB}} e_3$. First observe that no matter what u (i.e., lock/thread) and first^u_C (i.e., $\mathtt{acq}(\cdot)$, event, or $\mathtt{join}(\cdot)$) are, we have $u \in \mathsf{Before}_C(e_2)$ because of first^u_C. Next, no matter what u and e_3 are, $e_3 \leq^B_{\mathsf{HB}} \mathsf{last}^u_B$. Thus, $e_1 \leq^B_{\mathsf{HB}} \mathsf{last}^u_B$ which establishes the claim.

We complete the proof by showing that (3) $\Rightarrow$ (1). Suppose there is $u \in \mathsf{Before}_C(e_2)$ such that last^u_B is defined and $e_1 \leq^B_{\mathsf{HB}} \mathsf{last}^u_B$. Let $e_3 = \mathsf{last}^u_B$. We have $e_1 \leq^B_{\mathsf{HB}} e_3$. Let $e_4 \in [\![C]\!]$ be the event that witnesses $u \in \mathsf{Before}_C(e_2)$; thus, $e_4 \leq^C_{\mathsf{HB}} e_2$. Since $\leq^B_{\mathsf{HB}} \subseteq \leq^A_{\mathsf{HB}}$ and $\leq^C_{\mathsf{HB}} \subseteq \leq^A_{\mathsf{HB}}$, we can conclude that $e_1 \leq^A_{\mathsf{HB}} e_3$ and $e_4 \leq^A_{\mathsf{HB}} e_2$. Finally, since $\leq^A_{\mathsf{HB}}$ is a transitive relation, it would be enough to establish that $e_3 \leq^A_{\mathsf{HB}} e_4$ in order to complete the proof. Considering the various possibilities for u, e_3, and e_4, we will have four sub-cases like in the proof of (1) $\Rightarrow$ (2), and in all cases we can conclude that $e_3 \leq^A_{\mathsf{HB}} e_4$. □

Now let us prove Lemma 3.5:

Lemma 3.5. *Let A be a non-terminal with rule $A \to BC$ and let $e \in [\![B]\!]$. Then*

$$\mathsf{After}_A(e) = \mathsf{After}_B(e) \cup \bigcup_{u \in \mathsf{After}_B(e)} \mathsf{AF}_C(u)$$

Proof. We need to prove that for a non-terminal A with rule $A \to BC$ and $e \in [\![B]\!]$,

$$\mathsf{After}_A(e) = \mathsf{After}_B(e) \cup \bigcup_{u \in \mathsf{After}_B(e)} \mathsf{AF}_C(u)$$

We prove containment in each direction. Consider $u' \in \mathsf{After}_A(e)$. Let us assume that event e' "witnesses" the membership of u' in the set $\mathsf{After}_A(e)$. If $e' \in [\![B]\!]$ then $u' \in \mathsf{After}_B(e)$ because of e'. On the other hand, if $e' \in [\![C]\!]$, then $e \leq^A_{\mathsf{HB}} e'$ and then by Lemma B.3, there is a $u \in \mathsf{After}_B(e)$ such that $e' \in \{e \in [\![C]\!] \mid \mathsf{first}^u_C \leq^C_{\mathsf{HB}} e\}$, where first^u_C is defined as in Lemma B.3. Thus, $u' \in \mathsf{After}_C(\mathsf{first}^u_C) = \mathsf{AF}_C(u)$, This establishes the containment from left to right.

Let us now consider the other direction. Observe that $\mathsf{After}_B(e) \subseteq \mathsf{After}_A(e)$. Consider $u' \in \cup_{u \in \mathsf{After}_B(e)} \mathsf{AF}_C(u)$. Let u_1 be such that $u_1 \in \mathsf{After}_B(e)$ and $u' \in \mathsf{AF}_C(u_1)$. Further, let e' be the event witnessing the membership of u' in $\mathsf{AF}_C(u_1)$. By Lemma B.3, $e \leq^A_{\mathsf{HB}} e'$ and therefore, $u' \in \mathsf{After}_A(e)$ which establishes the containment from right to left. □

Now consider the dual Lemma 3.6.

Lemma 3.6. *Let A be a non-terminal with rule $A \to BC$ and let $e \in [\![C]\!]$. Then*

$$\mathsf{Before}_A(e) = \mathsf{Before}_C(e) \cup \bigcup_{u \in \mathsf{Before}_C(e)} \mathsf{BL}_B(u)$$

Proof. The proof is similar to the proof of Lemma 3.5 and is therefore skipped. □

Complexity Analysis.

Theorem B.4. *Let G be an SLP of size g representing a string σ, having T threads, L locks, and V variables. The inductive algorithm described in this section to detect* HB*-races runs in time $O(g((T+L)^2(L+TV)))$ and uses space $O(g((T+L)(L+TV)))$.*

Proof. Let us begin with the space requirements of the algorithm. For each non-terminal A, the algorithm maintains a bit $\mathsf{Race?}(A)$, the sets $\mathsf{AF}_A(u)$ and $\mathsf{BL}_A(u)$ for each lock and thread u, and the collections $\mathsf{ALRd}_A(t,x)$, $\mathsf{ALWr}_A(x)$, $\mathsf{BFRd}_A(t,x)$, and $\mathsf{BFWr}_A(x)$ for each variable x and thread t. The sets $\mathsf{AF}_A(u)$ and $\mathsf{BL}_A(u)$ have at most $T+L$ elements. Each of the sets $\mathsf{ALRd}_A(t,x)$, $\mathsf{ALWr}_A(x)$, $\mathsf{BFRd}_A(t,x)$ and $\mathsf{BFWr}_A(x)$ have size at most $T+L$. Putting all of this together, the total space used is $O(g(1+(T+L)(T+L)+V(T+L)+(TV)(T+L))) = O(g(T+L)(T+L+V+TV)) = O(g((T+L)(L+TV)))$.

Let us now analyze the time requirements. The algorithm computes the various sets it maintains for each non-terminal. The time to compute $\mathsf{AF}_A(u)$ (or $\mathsf{BL}_A(u)$) is $O((L+T)(L+T))$, because it involves taking the union of at most $(L+T+1)$ AF (or BL) sets, each of which has size at most $L+T$. Next, let us analyze the time to compute $\mathsf{ALRd}_A(t,x)$ set; the time for $\mathsf{ALWr}_A(x)$, $\mathsf{BFRd}_A(t,x)$, and $\mathsf{BFWr}_A(x)$ is the same. Let us recall the inductive definition of $\mathsf{ALRd}_A(t,x)$ for the rule $A \to BC$.

$$\mathsf{ALRd}_A(t,x) = \begin{cases} \mathsf{ALRd}_C(t,x) & \text{if } \mathsf{ALRd}_C(t,x) \neq \emptyset \\ \mathsf{ALRd}_B(t,x) \cup \bigcup_{u \in \mathsf{ALRd}_B(t,x)} \mathsf{AF}_C(u) & \text{otherwise} \end{cases} \tag{14}$$

In the worst case, when $\mathsf{ALRd}_C(t,x)$ is $\emptyset$, one has to take the union of a maximum of $T+L+1$ sets, each of whose size is bounded by $T+L$. So the time to compute $\mathsf{ALRd}_A(t,x)$ is $O((T+L)(T+L))$.

Finally, let us analyze the time to compute Race?(A). From Equation (2), this requires checking the disjointness of sets of size at most $L+T$. This takes time $O(L+T)$.

All these observations together give a total running time of the entire algorithm to be

$$O(g[(T+L)(T+L)^2 + V(T+L)^2 + TV(T+L)^2 + V(T+L) + TV(T+L)]) = O(g((T+L)^2(L+TV)))$$

□

C PROOFS FROM SECTION 4

Proof of Lemma 4.1.

Before we present the proof of Lemma 4.1, it will be useful to formally state the definition of the function LocksHeld.

*Definition C.1 (*LocksHeld*).* For a trace σ, and an event $e = \langle t : o\rangle \in \sigma$, the set of locks held by thread t in trace σ when e is being performed, is defined as

$$\begin{aligned}\mathsf{LocksHeld}_\sigma(e) = &\{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathsf{acq}(\ell)\rangle \in \sigma \\ &\quad \text{such that } e' \leq^\sigma_{\mathsf{TO}} e \text{ and } \neg(\mathsf{match}(e') \leq^\sigma_{\mathsf{TO}} e)\ \} \\ &\cup \{\ell \in \mathsf{Locks}(\sigma) \mid \exists e' = \langle t : \mathtt{rel}(\ell)\rangle \in \sigma \\ &\quad \text{such that } e \leq^\sigma_{\mathsf{TO}} e' \text{ and } \neg(e \leq^\sigma_{\mathsf{TO}} \mathsf{match}(e'))\ \}\end{aligned}$$

Let us now present the proof of Lemma 4.1

Lemma 4.1. *Let A be a non-terminal with rule $A \to BC$. Let $e \in B\restriction_t$ and $e' \in C\restriction_{t'}$ be read/write events performed by threads t, t'. Then,*

$$\begin{aligned}\mathsf{LocksHeld}_A(e) = &\ \mathsf{LocksHeld}_B(e) \\ &\cup\{\ell \mid \mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)\} \\ \mathsf{LocksHeld}_A(e') = &\ \mathsf{LocksHeld}_C(e') \\ &\cup\{\ell \mid \mathsf{OpenAcq}_B(t',\ell) > \mathsf{OpenRel}_C(t',\ell)\}\end{aligned}$$

Proof. We will prove the first equation. The proof is similar for the second equation.

We have event $e \in B\restriction_t$. Let us first prove the containment

$$\begin{aligned}\mathsf{LocksHeld}_A(e) \subseteq &\ \mathsf{LocksHeld}_B(e) \\ &\cup\{\ell \mid \mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)\}\end{aligned}$$

Let $\ell \in \mathsf{LocksHeld}_A(e)$. Then, one of the following cases is possible.

(1) there is an $e' = \langle t : \mathsf{acq}(\ell)\rangle \in [\![A]\!]$ such that $e' \leq^A_{\mathsf{TO}} e$ and e' is either not released at all in A, or $\mathsf{match}e' \in C\restriction_t$. In either case, $\ell \in \mathsf{LocksHeld}_B(e)$.
(2) there is an $e' = \langle t : \mathtt{rel}(\ell)\rangle \in [\![A]\!]$ such that $e \leq^A_{\mathsf{TO}} e'$, and either $\mathsf{match}e'$ is not in A or is acquired in $B\restriction_t$. In the first case when e' is unmatched in A, we have $\mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)$. Otherwise, $\ell \in \mathsf{LocksHeld}_B(e)$.

In either case, ℓ is contained in the set on the right.

Now, let us prove the following containment:

$$\begin{aligned}\mathsf{LocksHeld}_A(e) \supseteq &\ \mathsf{LocksHeld}_B(e) \\ &\cup\{\ell \mid \mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)\}\end{aligned}$$

First, consider $\ell \in \mathsf{LocksHeld}_B(e)$. Clearly, $\ell \in \mathsf{LocksHeld}_A(e)$. Otherwise, consider ℓ such that $\mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)$. Thus, there is atleast one release event $e' = \langle t : \mathtt{rel}(\ell)\rangle \in [\![C]\!]$ such that $\mathsf{match}e' \notin [\![A]\!]$. In this case again, $\ell \in \mathsf{LocksHeld}_A(e)$. □

Complexity Analysis.

Theorem C.2. *Let G be an SLP of size g representing a string σ, having T threads, L locks, and V variables. Let r be the maximum number of times a thread acquires a lock without releasing it, in the trace generated by G. The inductive algorithm described in this section to detect violations of lockset discipline runs in time $O(gTL(\log r + V))$ and uses space $O(gTL(\log r + V))$.*

Proof. Let us first analyze the space complexity. For every non-terminal, we need to store $\mathsf{LockSet}(t,x)$ having size atmost L, for every pair of thread t and variable x. For every non-terminal, we also need to store integers $\mathsf{OpenAcq}(t,\ell)$ and $\mathsf{OpenAcq}(t,\ell)$, for every pair of lock ℓ and thread t. The size of every such integer is bounded by $O(\log r)$. The total space usage per non-terminal therefore is $O(TLV + TL\log r)$, and the overall space requirement is $O(gTL(V + \log r))$.

Let us now evaluate the time complexity. For every non-terminal, the algorithm detects lockset violation by checking if for some variable x, the intersection of atmost T sets, each of size L is empty. This takes time $O(TLV)$ per non-terminal. To compute LockSet, the algorithm, for every thread t and variable x performs set operations on constantly many sets of size $O(L)$. This takes time $O(TLV)$ for every non-terminal. Lastly, to compute OpenAcq and OpenRel, the algorithm performs arithmetic operations on constantly many integers (of size $O(\log r)$) for every thread and lock, taking time $O(TL\log r)$ per non-terminal. The overall time complexity therefore is $O(gTL(V + \log r))$. □

D COMPLETE ALGORITHM FOR HAPPENS BEFORE WITH PROOF OF CORRECTNESS

Here, we will first present the complete algorithm for detecting HB races, and then argue about correctness.

D.1 Algorithm

We present the complete inductive algorithm for detecting HB-races on traces represented by SLPs. The algorithm is inductive, and computes various sets for each non-terminal in the SLP. The definitions of the various sets being computed is given in Section 3. Here we only present the algorithms that compute these sets. Each of these sets is computed for a non-terminal A and the computation depends on the rule associated with A. Recall that there are two types of rules — either $A \to a$, where a is an event, or $A \to BC$, where B and C are non-terminals. In what follows, t and t' will be used to denote threads, ℓ to denote a lock, and u and u' to denote either a lock or thread.

Race predicate. For rule $A \to a$, Race?(A) = false. For $A \to BC$,

$$\begin{aligned}\mathsf{Race?}(A) = &\ \mathsf{Race?}(B) \vee \mathsf{Race?}(C) \vee \\ \bigvee_{x\in \mathsf{Wr}(B)\cap \mathsf{Wr}(C)} &\ \mathsf{ALWr}_B(x) \cap \mathsf{BFWr}_C(x) = \emptyset \\ \bigvee_{x\in \mathsf{Wr}(B),(t,x)\in \mathsf{Rd}(C)} &\ \mathsf{ALWr}_B(x) \cap \mathsf{BFRd}_C(t,x) = \emptyset \\ \bigvee_{(t,x)\in \mathsf{Rd}(B),x\in \mathsf{Wr}(C)} &\ \mathsf{ALRd}_B(t,x) \cap \mathsf{BFWr}_C(x) = \emptyset\end{aligned}$$

After First Sets. For rule $A \to a$, we have

$$\mathsf{AF}_A(t) = \begin{cases} \{t, \ell\} & \text{if } a = \langle t : \mathtt{rel}(\ell)\rangle \\ \{t, t'\} & \text{if } a = \langle t : \mathtt{fork}(t')\rangle \\ \{t\} & \text{if } a = \langle t : o\rangle \text{ and } o \notin \{\mathtt{rel}(\cdot), \mathtt{fork}(\cdot)\} \\ \{t'\} & \text{if } a = \langle t' : \mathtt{join}(t)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{AF}_A(\ell) = \begin{cases} \{t\} & \text{if } a = \langle t : \mathtt{acq}(\ell)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

For rule $A \to BC$ (and u either a thread t or lock ℓ)

$$\mathsf{AF}_A(u) = \mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u')$$

Before Last Sets. For rule $A \to a$,

$$\mathsf{BL}_A(t) = \begin{cases} \{t, \ell\} & \text{if } a = \langle t : \mathtt{acq}(\ell)\rangle \\ \{t, t'\} & \text{if } a = \langle t : \mathtt{join}(t')\rangle \\ \{t\} & \text{if } a = \langle t : o\rangle \text{ and } o \notin \{\mathtt{acq}(\cdot), \mathtt{join}(\cdot)\} \\ \{t'\} & \text{if } a = \langle t' : \mathtt{fork}(t)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{BL}_A(\ell) = \begin{cases} \{t\} & \text{if } a = \langle t : \mathtt{rel}(\ell)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

For rule $A \to BC$

$$\mathsf{BL}_A(u) = \mathsf{BL}_C(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{BL}_C(u)} \mathsf{BL}_B(u')$$

After Read Sets. For rule $A \to a$,

$$\mathsf{ALRd}_A(t, x) = \begin{cases} \{t\}, & \text{if } a = \langle t : \mathtt{r}(x)\rangle \\ \emptyset, & \text{otherwise} \end{cases}$$

For rule $A \to BC$,

$$\mathsf{ALRd}_A(t, x) = \begin{cases} \mathsf{ALRd}_C(t, x) & \text{if } \mathsf{ALRd}_C(t, x) \neq \emptyset \\ \mathsf{ALRd}_B(t, x) \cup \bigcup_{u \in \mathsf{ALRd}_B(t,x)} \mathsf{AF}_C(u) & \text{otherwise} \end{cases}$$

After Write Sets. For rule $A \to a$,

$$\mathsf{ALWr}_A(x) = \begin{cases} \{t\} & \text{if } = \langle t : \mathtt{w}(x)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

For rule $A \to BC$,

$$\mathsf{ALWr}_A(x) = \begin{cases} \mathsf{ALWr}_C(x) & \text{if } \mathsf{ALWr}_C(x) \neq \emptyset \\ \mathsf{ALWr}_B(x) \cup \bigcup_{u \in \mathsf{ALWr}_B(x)} \mathsf{AF}_C(u) & \text{otherwise} \end{cases}$$

Before Read Sets. For rule $A \to a$,

$$\mathsf{BFRd}_A(t, x) = \begin{cases} \{t\}, & \text{if } a = \langle t : \mathtt{r}(x)\rangle \\ \emptyset, & \text{otherwise} \end{cases}$$

For rule $A \to BC$,

$$\mathsf{BFRd}_A(t, x) = \begin{cases} \mathsf{BFRd}_B(t, x) & \text{if } \mathsf{BFRd}_B(t, x) \neq \emptyset \\ \mathsf{BFRd}_C(t, x) \cup \bigcup_{u \in \mathsf{BFRd}_C(t,x)} \mathsf{BL}_B(u) & \text{otherwise} \end{cases}$$

Before Write Sets. For rule $A \to a$,

$$\mathsf{BFWr}_A(x) = \begin{cases} \{t\} & \text{if } = \langle t : \mathtt{w}(x)\rangle \\ \emptyset & \text{otherwise} \end{cases}$$

For rule $A \to BC$,

$$\mathsf{BFWr}_A(x) = \begin{cases} \mathsf{BFWr}_B(x) & \text{if } \mathsf{BFWr}_B(x) \neq \emptyset \\ \mathsf{BFWr}_C(x) \cup \bigcup_{u \in \mathsf{BFWr}_C(x)} \mathsf{BL}_B(u) & \text{otherwise} \end{cases}$$

D.2 Correctness of HB algorithm

In this section, we prove that the inductive algorithm for HB described in the previous section is correct. The proof will rely on Lemmas 3.5 and 3.6, and Theorem 3.3.

Theorem D.1. *The algorithm presented in the Appendix correctly detects* HB*-races.*

Proof. Theorem 3.3 already establishes the correctness of the definition of the predicate Race?(·). Thus, to complete the proof of the correctness of our inductive algorithm, all we need to prove is that the sets AF, BL, ALRd, ALWr, BFRd, and BFWr are all being computed correctly by the inductive definitions in the Appendix. We will argue the correctness of the sets AF and ALRd; the proof of correctness for the remaining cases is similar and skipped.

Let us begin the computation of $\mathsf{AF}_A(u)$ for a non-terminal A and lock/thread u. When A has a rule $A \to a$, where a is an event, the computation of AF_A is consistent with the definition in Equation (3) and Equation (4) . In the inductive step, consider A with rule $A \to BC$. We need to prove that

$$\mathsf{AF}_A(u) = \mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u')$$

Let us begin by considering the simple case when first_A^u = undefined. In this case, by definition, $\mathsf{AF}_A(u) = \emptyset$. Also, since first_A^u = undefined, first_B^u = undefined and first_C^u = undefined. Therefore, inductively, $\mathsf{AF}_B(u) = \emptyset$ and $\mathsf{AF}_C(u) = \emptyset$, giving us that the right hand side will also evaluate to $\emptyset$. So let us now consider the case where $\mathsf{first}_A^u = e \in [\![A]\!]$ is defined. There are two possibilities to consider. If $e \in [\![C]\!]$ (and therefore $e \notin [\![B]\!]$) then $\mathsf{AF}_B(u) = \emptyset$, and

$$\mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u') = \emptyset \cup \mathsf{AF}_C(u) = \mathsf{AF}_A(u)$$

which is indeed correct. The second sub-case to consider is when $e \in [\![B]\!]$. Then, by Lemma 3.5, we have

$$\begin{aligned}\mathsf{AF}_A(u) = \mathsf{After}_A(e) &= \mathsf{After}_B(e) \cup \bigcup_{u' \in \mathsf{After}_B(e)} \mathsf{AF}_C(u') \\ &= \mathsf{AF}_B(u) \cup \bigcup_{u' \in \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u')\end{aligned}$$

Since $u \in \mathsf{AF}_B(u)$, we have $\mathsf{AF}_B(u) = \{u\} \cup \mathsf{AF}_B(u)$, and thus,

$$\mathsf{AF}_A(u) = \mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u')$$

Let us now establish the correctness of the inductive definition of $\mathsf{ALRd}_A(t,x)$ for a non-terminal A and variable x. Again the base case of the non-terminal having rule $A \to a$, where a is an event, is clearly correct as per the definition in Equation (1). In the inductive step of a rule $A \to BC$, we need to prove the correctness of the following equation:

$$\mathsf{ALRd}_A(t,x) = \begin{cases} \mathsf{ALRd}_C(t,x) & \text{if } \mathsf{ALRd}_C(t,x) \neq \emptyset \\ \mathsf{ALRd}_B(t,x) \cup \bigcup_{u \in \mathsf{ALRd}_B(t,x)} \mathsf{AF}_C(u) & \text{otherwise} \end{cases}$$

Let $e = \mathsf{Last}_A(\mathsf{REvents}_A(t,x))$. If e is undefined, then both $\mathsf{ALRd}_C(t,x)$ and $\mathsf{ALRd}_B(t,x)$ are $\emptyset$, and so is $\mathsf{ALRd}_A(t,x)$.

Next consider the case when e is defined and belongs to the chunk C (i.e., $e \in [\![C]\!]$). Then, clearly, $\mathsf{After}_A(e) = \mathsf{After}_C(e) = \mathsf{ALRd}_C(t,x)$. Inductively, $\mathsf{ALRd}_C(t,x)$ will indeed by non-empty and thus $\mathsf{ALRd}_A(t,x) = \mathsf{ALRd}_C(t,x)$.

The last case is when $e \in [\![B]\!]$. The correctness then follows from Lemma 3.5. □

E COMPLETE ALGORITHM FOR LOCKSET COMPUTATION ON COMPRESSED TRACES WITH PROOF OF CORRECTNESS

Here, we will first present the complete algorithm for detecting violations of the lockset discipline, and then argue about correctness.

E.1 Algorithm

We present the complete inductive algorithm for detecting violations of lockset discipline. on traces represented by SLPs. The algorithm is inductive, and computes various data-structures for each non-terminal in the SLP. The definitions of OpenAcq and OpenRel is given in Section 4. Here we will first precisely state the definition for LockSet:

Definition E.1. For a trace σ, thread t and variable x, we define

$$\mathsf{LockSet}_\sigma(t,x) = \begin{cases} \top & \text{if } \mathsf{Access}_\sigma(t,x) = \emptyset \\ \mathsf{RealLocks}_\sigma^{(t,x)} \cup \{\Lambda, \Lambda_t\} & \text{if } x \notin \mathsf{Wr}(\sigma\restriction_t) \\ \mathsf{RealLocks}_\sigma^{(t,x)} \cup \{\Lambda_t\} & \text{otherwise} \end{cases}$$

where

$$\mathsf{RealLocks}_\sigma^{(t,x)} = \bigcap_{e \in \mathsf{Access}_\sigma(t,x)} \mathsf{LocksHeld}_\sigma(t,x)$$

The set $\top$ in the above definition, is a universal set such that for any set S, $S \cap \top = S$, and $S \cup \top = \top$. Also, $\top \cap \top = \top$ and $\top \cup \top = \top$. This is introduced only for notational convenience. When the set of all locks (including dummy locks) is known, it can be used instead of using $\top$.

We will now present the algorithms that compute these data. Each of these data-structures is computed for a non-terminal A and the computation depends on the rule associated with A. Recall that there are two types of rules — either $A \to a$, where a is an event, or $A \to BC$, where B and C are non-terminals.

Computing $\mathsf{OpenAcq}_A$.

For rule $A \to a$,

$$\mathsf{OpenAcq}_A(t,\ell) = \begin{cases} 1, & \text{if } a = \langle t : \mathsf{acq}(\ell) \rangle \\ 0, & \text{otherwise} \end{cases}$$

For $A \to BC$,

$$\begin{aligned}\mathsf{OpenAcq}_A(t,\ell) = \ & \mathsf{OpenAcq}_C(t,\ell) \\ & + \max\{0, \mathsf{OpenAcq}_B(t,\ell) - \mathsf{OpenRel}_C(t,\ell)\}\end{aligned}$$

Computing $\mathsf{OpenRel}_A$.

For rule $A \to a$,

$$\mathsf{OpenRel}_A(t,\ell) = \begin{cases} 1, & \text{if } a = \langle t : \mathtt{rel}(\ell) \rangle \\ 0, & \text{otherwise} \end{cases}$$

For $A \to BC$,

$$\begin{aligned}\mathsf{OpenRel}_A(t,\ell) = \ & \mathsf{OpenRel}_B(t,\ell) \\ & + \max\{0, \mathsf{OpenRel}_C(t,\ell) - \mathsf{OpenAcq}_B(t,\ell)\}\end{aligned}$$

Computing $\mathsf{LockSet}_A$.

For rule $A \to a$,

$$\mathsf{LockSet}_A(t,\ell) = \begin{cases} \{\Lambda, \Lambda_t\}, & \text{if } a = \langle t : \mathtt{r}(x) \rangle \\ \{\Lambda_t\}, & \text{if } a = \langle t : \mathtt{w}(x) \rangle \\ \top, & \text{otherwise} \end{cases}$$

For $A \to BC$,

$$\begin{aligned}&\mathsf{LockSet}_A(t,x) \\ &= \Big(\mathsf{LockSet}_B(t,x) \cup \{\ell \mid \mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)\}\Big) \\ &\cap \Big(\mathsf{LockSet}_C(t,x) \cup \{\ell \mid \mathsf{OpenAcq}_B(t',\ell) > \mathsf{OpenRel}_C(t',\ell)\}\Big)\end{aligned}$$

Checking lockset violation.

For a non-terminal A, violation is reported if there is a variable x accessed in A for which the following condition holds

$$\bigcap_{t \in \mathsf{Threads}(A)} \mathsf{LockSet}_A(t,x) = \emptyset$$

E.2 Correctness of LockSet algorithm

Here, we will prove that the inductive algorithm presented in Section E.1 is correct.

We first note the following simple but important observation, which is similar to Proposition B.2, and informally states that, the presence/absence of lockset violations in a trace σ, does not get affected by the "context" in which σ is placed.

Proposition E.2. *Let σ be a trace and x be a variable accessed in σ. The following two statements are equivalent*

(1)

$$\bigcap_{t \in \text{Threads}(\sigma)} \text{LockSet}_\sigma(t,x) = \emptyset$$

(2) For any trace $\sigma' = \sigma_1\sigma\sigma_2$,

$$\bigcap_{t \in \text{Threads}(\sigma')} \text{LockSet}_{\sigma'}(t,x) = \emptyset$$

Thus if a non-terminal different from the start symbol S of the grammar G reports a violation, there will be a violation in S too. Finally, the correctness of the inductive algorithm will be complete if we prove that each of the inductive formulations in the previous section are correct:

Theorem E.3. *The algorithm presented in Section E.1 correctly detects violations of lockset discipline.*

Proof. We will prove that all the inductive formulations are correct.

First, let us prove the correctness of the inductive formulation for $\text{OpenAcq}_A(t,\ell)$. We will skip the proof for $\text{OpenRel}_A(t,\ell)$ because it is similar.

Let us fix some notation before we begin. For non-terminal D, thread t and lock ℓ, let

$$\text{unmatchedAcq}_D^{(t,\ell)} = \{e = \langle t : \text{acq}(\ell)\rangle \in D \mid \text{match}(e) \notin [\![D]\!]\}.$$

and

$$\text{unmatchedRel}_D^{(t,\ell)} = \{e = \langle t : \texttt{rel}(\ell)\rangle \in D \mid \text{match}(e) \notin [\![D]\!]\}.$$

Then, $\text{OpenAcq}_D(t,\ell) = |\text{unmatchedAcq}_D^{(t,\ell)}|$ and $\text{OpenRel}_D(t,\ell) = |\text{unmatchedRel}_D^{(t,\ell)}|$.

In the base case, we have the rule $A \to a$. The set $\text{unmatchedAcq}_A^{(t,\ell)}$ is non-empty only if $a = \langle t : \text{acq}(\ell)\rangle$ is an acquire event in which case $|\text{unmatchedAcq}_A^{(t,\ell)}| = 1$. In all other cases $|\text{unmatchedAcq}_A^{(t,\ell)}| = 0$. This justifies the base case.

In the inductive case, we have the rule $A \to BC$. Then, the set $\text{unmatchedAcq}_A^{(t,\ell)}$ can be expressed as a disjoint union $S_B \cup S_C$ ($S_B \cap S_C = \emptyset$), where

$$S_B = \{e \in \text{unmatchedAcq}_A^{(t,\ell)} \mid e \in B\restriction_t\}$$

and

$$S_C = \{e \in \text{unmatchedAcq}_A^{(t,\ell)} \mid e \in C\restriction_t\}$$

First, note that for every event $e \in S_C$, match$e \notin A$ and thus match$e \notin C$. Thus we have $S_C \subseteq \text{unmatchedAcq}_C^{(t,\ell)}$. Next, see that every event $e \in \text{unmatchedAcq}_C^{(t,\ell)}$ is clearly unmatched even in A, and thus belongs to $\text{unmatchedAcq}_A^{(t,\ell)}$, and thus to S_C. This gives $\text{unmatchedAcq}_C^{(t,\ell)} \subseteq S_C$. Hence, $S_C = \text{unmatchedAcq}_C^{(t,\ell)}$. The cardinality of $S_C = \text{unmatchedAcq}_C^{(t,\ell)} = \text{OpenAcq}_C(t,\ell)$. Next, consider the following set

$$S_B^1 = \{e \in \text{unmatchedAcq}_B^{(t,\ell)} \mid \text{match}(e) \in \text{unmatchedRel}_C^{(t,\ell)}\}$$

Also, let $S_B^2 = \text{unmatchedAcq}_B^{(t,\ell)} \setminus S_B^1$. Then, every event $e \in S_B^2$ is unmatched in A, and thus belongs to $\text{unmatchedAcq}_A^{(t,\ell)}$ and hence to $\text{unmatchedAcq}_B^{(t,\ell)}$. This means $S_B^2 \subseteq S_B$. Also, consider an event $e \in S_B$. Since it is unmatched in A, it must also be unmatched in B, and thus $e \in \text{unmatchedAcq}_B^{(t,\ell)}$. Also, it does not have a matching release in C and thus, $e \notin S_B^1$. Thus, $e \in S_B^2$. Clearly, $S_B = S_B^2$. Also, the cardinality of S_B^2 is just $\max\{0, \text{OpenAcq}_B(t,\ell) - \text{OpenRel}_C(t,\ell)\}$.

Since $S_B \cap S_C = \emptyset$, we have the desired result.

Let us now prove the correctness of the inductive formulation of LockSet. The base case $A \to a$ indeed matches Definition E.1.

We consider the case $A \to BC$. If $\text{Access}_A(t,x) = \emptyset$, both $\text{Access}_B(t,x) = \emptyset$ and $\text{Access}_C(t,x) = \emptyset$. In this case, using the inductive hypothesis, $\text{LockSet}_B(t,x) = \top$ and $\text{LockSet}_C(t,x) = \top$. Thus, $\text{LockSet}_A(t,x)$ evaluates to $\top$, which is correct.

Now consider the case when $\text{Access}_A(t,x)$ is non-empty. If $\text{Access}_B(t,x)$ is empty, then inductively, $\text{LockSet}_B(t,x) = \top$ and $\text{LockSet}_A(t,x)$ evaluates to $\text{LockSet}_C(t,x)$. This is correct since $\text{Access}_A(t,x) = \text{Access}_C(t,x)$ and $\text{Wr}(A\restriction_t) = \text{Wr}(C\restriction_t)$ in this case. The case when $\text{Access}_C(t,x)$ is similar. Next, consider the case when $\text{Access}_B(t,x) \neq \emptyset$ and $\text{Access}_C(t,x) \neq \emptyset$. By the inductive hypothesis, $\Lambda_t \in \text{LockSet}_B(t,x)$ and $\Lambda_t \in \text{LockSet}_C(t,x)$. Clearly, $\Lambda_t \in \text{LockSet}_A(t,x)$, as expected. Now we have two cases. First, the case when $\text{Access}_A(t,x) = \text{REvents}_A(t,x)$, that is, there is no write event to x in $A\restriction_t$. Then, $\text{Access}_B(t,x) = \text{REvents}_B(t,x)$ and $\text{Access}_C(t,x) = \text{REvents}_C(t,x)$. By induction, $\Lambda \in \text{LockSet}_B(t,x)$ and $\Lambda \in \text{LockSet}_C(t,x)$. Thus, $\Lambda \in \text{LockSet}_A(t,x)$, which is correct. Otherwise, atlast one of $\text{LockSet}_B(t,x)$ and $\text{LockSet}_C(t,x)$ does not have Λ. As a result, $\Lambda \notin \text{LockSet}_A(t,x)$ Finally,

$$\begin{aligned}\text{RealLocks}_A^{(t,x)} &= \bigcap_{e \in \text{Access}_A(t,x)} \text{LocksHeld}_A(e)\\ &= \bigcap_{e \in \text{Access}_B(t,x)} \text{LocksHeld}_A(e) \cap \bigcap_{e \in \text{Access}_C(t,x)} \text{LocksHeld}_A(e)\end{aligned}$$

By Lemma 4.1, for an event $e \in B$, we have

$$\begin{aligned}\text{LocksHeld}_A(e) =& \text{LocksHeld}_B(e)\\ &\cup \{\ell \mid \text{OpenRel}_C(t,\ell) > \text{OpenAcq}_B(t,\ell)\}\end{aligned}$$

Thus,

$$\begin{aligned}\bigcap_{e \in \text{Access}_B(t,x)} \text{LocksHeld}_A(e) =& \bigcap_{e \in \text{Access}_B(t,x)} \Big(\text{LocksHeld}_B(e) \cup\\ &\{\ell \mid \text{OpenRel}_C(t,\ell) > \text{OpenAcq}_B(t,\ell)\}\Big)\\ =& \Big(\bigcap_{e \in \text{Access}_B(t,x)} \text{LocksHeld}_B(e)\Big)\\ &\cup \{\ell \mid \text{OpenRel}_C(t,\ell) > \text{OpenAcq}_B(t,\ell)\}\\ =& \text{RealLocks}_B^{(t,x)} \cup\\ &\{\ell \mid \text{OpenRel}_C(t,\ell) > \text{OpenAcq}_B(t,\ell)\}\end{aligned}$$

Similarly,

$$\begin{aligned}\bigcap_{e \in \text{Access}_C(t,x)} \text{LocksHeld}_A(e) =& \text{RealLocks}_C^{(t,x)} \cup\\ &\{\ell \mid \text{OpenAcq}_B(t,\ell) > \text{OpenRel}_B(t,\ell)\}\end{aligned}$$

Thus, we have

$$\begin{aligned}&\mathsf{LockSet}_A(t,x)\\&= \Big(\mathsf{LockSet}_B(t,x) \cup \{\ell \mid \mathsf{OpenRel}_C(t,\ell) > \mathsf{OpenAcq}_B(t,\ell)\}\Big)\\&\cap\Big(\mathsf{LockSet}_C(t,x) \cup \{\ell \mid \mathsf{OpenAcq}_B(t',\ell) > \mathsf{OpenRel}_C(t',\ell)\}\Big)\end{aligned}$$

□