Data Race Detection on Compressed Traces

Dileep Kini
Akuna Capital LLC
USA
dileeprkini@gmail.com

Umang Mathur
University of Illinois, Urbana Champaign
USA
umathur3@illinois.edu

Mahesh Viswanathan
University of Illinois, Urbana Champaign
USA
vmahesh@illinois.edu

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.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Formal software verification*;

KEYWORDS

Dynamic Program Analysis, Race Detection, Compression

ACM Reference Format:
Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 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 '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236025

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 [24, 48], graph based analysis [7, 22] or even vector clocks [20, 43]. 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 [26, 37].

However program traces are often huge, recording millions and billions of events. When 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 [14]. In such circumstances, the only way to alleviate the warehousing needs of storing such traces is to compress them [26, 37].

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 [18, 20, 24, 29, 36, 43, 49]. 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 [23], 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 [6, 11, 16, 23, 35, 42, 54]. 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 [32], and the lightweight lockset-based algorithm of Eraser [49] — 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.

[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* [20] in the race detection literature.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236025

```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

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 disk space. However, when this trace is compressed using the Sequitur algorithm [2, 40, 41], the compressed representation only occupies about 34 MB of disk space. The reason for this effective compression is that the program has a simple loop structure that is executed multiple times. Thus the program 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 is encountered. Compression makes this regular structure "visible", and an algorithm working on the compressed representation can exploit it by only performing an analysis only once for each distinct sub-trace.

We consider compression schemes that compress traces as straight line programs (SLPs). SLPs are a special class of context-free grammars 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 [62], 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, book-keeping 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 [12], where the book-keeping information 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 briefly present our happens-before based data race detection algorithm on compressed traces. Our algorithm for checking violations of the lockset discipline on compressed traces is presented briefly in Section 4. Details of these algorithms and their proof of correctness are presented in our companion technical report [30]. We present our experimental results in Section 5.

Related Work. Type systems to prevent data races have been developed [5, 9, 19]. Since the race detection problem is undecidable, the several static analysis techniques [13, 38, 39, 44, 46, 55, 58, 61] suffer from two problems — they don't scale and they raise many false alarms since they are typically conservative. Dynamic race detection techniques can be classified into three categories. There are the unsound lockset-based techniques, which may raise false alarms [49]. Techniques like random testing [50] and static escape analysis [45] can reduce the false alarms in such algorithms, but not eliminate them. The second category of dynamic analysis techniques are predictive runtime analysis techniques [24, 25, 34, 48, 56], 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 [32]. Early vector-clock based algorithms to compute happens-before on traces [18, 36] have been subsequently optimized [20, 43]. A lockset-based method for computing the happens-before partial order was proposed in [12]. Structured parallelism has been exploited to optimize the memory overhead in detecting happens before [10, 17, 47, 53, 60]. More recently, partial order that are weaker than happens before have been proposed for detecting data races, including causal precedence [51] and weak causal precedence [29]. Sofya [31] and RoadRunner [21] 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 a child thread is forked and joined by the same parent thread. Locks are assumed to be *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

[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. Even though 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	$\mathtt{w}(x)$	
2	$\mathtt{fork}(2)$	
3		$\mathtt{r}(x)$
4		$\mathtt{acq}(\ell)$
5		$\mathtt{w}(y)$
6		$\mathtt{rel}(\ell)$
7	$\mathtt{r}(x)$	
8	$\mathtt{acq}(\ell)$	
9	$\mathtt{rel}(\ell)$	
10	$\mathtt{w}(y)$	
11		$\mathtt{r}(x)$
12		$\mathtt{acq}(\ell)$
13		$\mathtt{w}(y)$
14		$\mathtt{rel}(\ell)$
15	$\mathtt{join}(2)$	
16	$\mathtt{w}(y)$	

$S \to AB$

$A \to CD$

$C \to EF$

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

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

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

$B \to FG$

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

Figure 2: Example trace σ_1 and its SLP representation

being acquired by some other thread. Therefore, with every release event $e = \langle t : \mathtt{rel}(\ell)\rangle$, we can associate a unique acquire event $e' = \langle t : \mathtt{acq}(\ell)\rangle$, which is the last $\mathtt{acq}(\ell)$-event in thread t before e that is not matched with any $\mathtt{rel}(\ell)$ event in thread t before e. This $\mathtt{acq}(\ell)$ event e' is said to be the matching acquire of e, and is denoted by $\mathsf{match}(e)$. Similarly, for an acquire event e' such that $e' = \mathsf{match}(e)$, we will say that e is the matching release of e', and we will also denote this by $\mathsf{match}(e')$. For a trace σ, $\sigma\lceil_t$ 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 $\mathsf{Locks}(\sigma)$. $\mathsf{Threads}(\sigma)$ 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

$$\begin{aligned}\mathsf{Threads}(\sigma) = \{t \mid \exists e \in \sigma.\ e = \langle t : o\rangle \text{ for some } o, \text{ or } e = \langle t' : \mathtt{fork}(t)\rangle \\ \text{or } e = \langle t' : \mathtt{join}(t)\rangle \text{ for some thread } t'\}.\end{aligned}$$

For a variable x, the set of $\mathtt{w}(x)$-events will be denoted by $\mathsf{WEvents}_\sigma(x)$ and the set of $\mathtt{r}(x)$-events performed by thread $t \in \mathsf{Threads}(\sigma)$ will be denoted by $\mathsf{REvents}_\sigma(t, x)$. We will use $\mathsf{Rd}(\sigma)$ to denote the set of pairs (t, x) for which $\mathsf{REvents}_\sigma(t, x) \neq \emptyset$. Similarly, we will use $\mathsf{Wr}(\sigma)$ to denote the set of variables x for which $\mathsf{WEvents}_\sigma(x)$ 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 $\mathsf{Last}_\sigma(S)$ the (unique) event $e \in S$, that is latest in σ among the events in S. Similarly, $\mathsf{First}_\sigma(S)$ is the event $e \in S$ that is earliest in σ 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 holds.

(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 $e_1 = \langle t_1 : \mathtt{a}_1(x)\rangle$ and $e_2 = \langle t_2 : \mathtt{a}_2(x)\rangle$ (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 $\mathtt{a}_1$ and $\mathtt{a}_2$ is $\mathtt{w}$. 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 [18, 20, 32, 36, 43] 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 [12] 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, $\mathsf{After}_{\sigma'}(\mathsf{undefined})$ 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. The lockset algorithm [49] is a low overhead technique to detect potential races. 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 $\mathcal{D} = \{\Lambda\} \cup \{\Lambda_t \,|\, t \text{ is a thread}\}$ 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 [43]. For a read/write event event $e = \langle t : \mathtt{a}(x)\rangle$ (where $\mathtt{a}$ is either $\mathtt{r}$ or $\mathtt{w}$) in trace σ, $\mathsf{LocksHeld}_\sigma(e)$ is the set of locks held by t when e is performed. Using this, for an event $e = \langle t : \mathtt{a}(x)\rangle$ we define $\mathsf{LockSet}_\sigma(e)$ to be

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

For a variables x and thread t, let $\mathsf{Access}_\sigma(t, x)$ be the set of all events in $\sigma\lceil_t$ whose corresponding operations are either $\mathtt{r}(x)$ or $\mathtt{w}(x)$. 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 $\mathsf{LocksHeld}_\sigma(e)$. To compute this for traces having reentrant locks, 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 briefly highlight the importance of the locks in $\mathcal{D}$ that were introduced. Let $\mathsf{LS}(x) = \cap_{t \in \mathsf{Threads}(\sigma)} \mathsf{LockSet}_\sigma(t, x)$. If the variable x is only accessed by a single thread t_1, then $\mathsf{LS}(x)$ is non-empty because it contains Λ_{t_1}. And if a variable x is only read and never written to, then $\mathsf{LS}(x)$ is again non-empty because it contains Λ. The Eraser algorithm [49] checks for violation of the lockset principle by maintaining the lockset for each thread-variable pair.

	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)	

$$S \to UV$$
$$U \to WX$$
$$W \to \langle 1 : \mathtt{r}(x)\rangle\langle 1 : \mathtt{acq}(\ell)\rangle$$
$$X \to \langle 1 : \mathtt{w}(y)\rangle\langle 1 : \mathtt{rel}(\ell)\rangle\langle 2 : \mathtt{acq}(\ell)\rangle$$
$$V \to YZ$$
$$Y \to \langle 2 : \mathtt{r}(x)\rangle\langle 2 : \mathtt{w}(y)\rangle$$
$$Z \to \langle 2 : \mathtt{rel}(\ell)\rangle\langle 2 : \mathtt{r}(x)\rangle\langle 1 : \mathtt{w}(z)\rangle$$

Figure 3: Example trace σ_2 and its SLP representation

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.

$$\begin{array}{ll} \mathsf{LockSet}_{\sigma_2}(1, x) = \{\Lambda, \Lambda_1\} & \mathsf{LockSet}_{\sigma_2}(2, x) = \{\Lambda, \Lambda_2\} \\ \mathsf{LockSet}_{\sigma_2}(1, y) = \{\Lambda_1, \ell\} & \mathsf{LockSet}_{\sigma_2}(2, y) = \{\Lambda_2, \ell\} \\ \mathsf{LockSet}_{\sigma_2}(1, z) = \{\Lambda_1\} & \mathsf{LockSet}_{\sigma_2}(2, z) = \{\Lambda, \Lambda_1, \Lambda_2, \ell\} \end{array}$$

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,

$$\begin{array}{ll} \mathsf{LockSet}_{\sigma_1}(1, x) = \{\Lambda_1\} & \mathsf{LockSet}_{\sigma_1}(2, x) = \{\Lambda, \Lambda_2\} \\ \mathsf{LockSet}_{\sigma_1}(1, y) = \{\Lambda_1\} & \mathsf{LockSet}_{\sigma_1}(2, y) = \{\Lambda_2, \ell\}. \end{array}$$

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 of 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 $\{A_{[i,i]} \mid 1 \leq i \leq n\} \cup \{A_{[1,i]} \mid 1 \leq i \leq n\}$ 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 $1 \leq i \leq n$. 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: $N = \{A_i \mid 0 \leq i \leq n\}$ with rules $A_0 \to a$, and $A_{i+1} \to A_iA_i$. 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 [40, 41]. 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 [59], LZ77 [62], LZW [57], Bisection [28], longest match [27] and Re-Pair [33].

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 algorithms analyzing uncompressed traces with running times at least linear

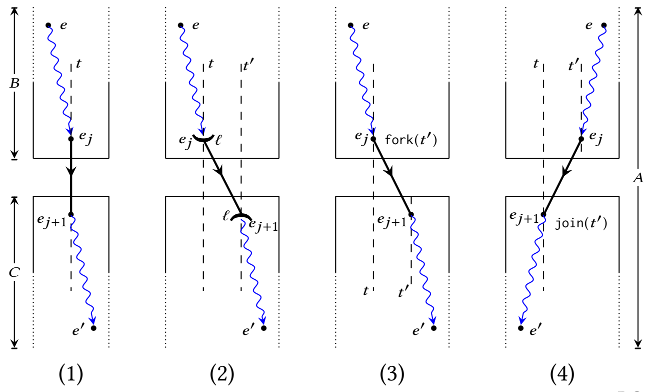


Figure 4: Illustrating the various scenarios that establish $e_j <^{BC}_{\mathsf{HB}} e_{j+1}$. In (2), ‘$\frown \ell$’ represents $\mathtt{acq}(\ell)$, ‘$\smile \ell$’ is $\mathtt{rel}(\ell)$.

in the size of the uncompressed trace). 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 the rule corresponding to A has the form $A \to BC$, 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 predicates $\mathsf{Race?}(B)$ 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.

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 or a common 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.

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

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 that 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*

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

and

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

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,

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

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,

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

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

Next we state the correctness of the Race? predicate.

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.

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

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 the 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 any $\mathsf{w}(x)$-event in E and any $\mathsf{r}(x)$-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 any pair of $\mathsf{w}(y)$-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 $\mathsf{r}(x)$-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 we can use Equation (2) to determine if a chunk has a race. 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 in [30].

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 [30]. 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 set $\mathsf{After}_A(e)$, where e is the last event amongst the read events $\mathsf{REvents}_A(t,x)$. If variable x is never read by thread t in the chunk A (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. In the first case, this last read event e belongs to the chunk $[\![C]\!]$. In this, clearly, $e = \mathsf{Last}_A(\mathsf{REvents}_A(t,x))$. 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 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 $\mathsf{ThEvents}^{\mathsf{join}}_D(t) = \{e \in D \mid e = \langle t : o\rangle \text{ or } e = \langle t' : \mathtt{join}(t)\rangle\}$.

Similarly, for a lock ℓ, define

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

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

We now formalize our intuitions in the following lemma.

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 in our technical report [30]. Its statement gives us the following inductive definition for $\mathsf{ALRd}_A(t,x)$.

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

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

To complete the formal definition 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 and is deferred to [30]. Consider the inductive step, of a non-terminal A with rule $A \to BC$ and let t be some thread. 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, $\mathsf{AF}_A(t) = \mathsf{AF}_C(t)$. A similar reasoning applies for a lock ℓ as well. Putting all these observations together, we get

$$\mathsf{AF}_A(u) = \mathsf{AF}_B(u) \cup \bigcup_{u' \in \{u\} \cup \mathsf{AF}_B(u)} \mathsf{AF}_C(u') \tag{6}$$

Let us now discuss 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 $[\![C]\!]$. In the former case, we have $\mathsf{Before}_A(e) = \mathsf{Before}_B(e)$. 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.

For a non-terminal D, thread t and lock ℓ,

$$\mathsf{BL}_D(t) = \mathsf{Before}_D(\mathsf{Last}_D(\mathsf{ThEvents}^{\mathsf{fork}}_D(t))) \tag{7}$$

$$\mathsf{BL}_D(\ell) = \mathsf{Before}_D(\mathsf{Last}_D(\mathsf{RelEvents}_D(\ell))) \tag{8}$$

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

The dual of Lemma 3.5 is the following lemma.

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 $\mathsf{BFRd}_A(t,x)$ is

$$\mathsf{BFRd}_A(t,x) = \begin{cases} \mathsf{BFRd}_B(t,x) & \text{if } \mathsf{BFRd}_B(t,x) \neq \emptyset \\ \mathsf{BFRd}_C(t,x) \cup & \text{otherwise} \\ \bigcup_{u \in \mathsf{BFRd}_C(t,x)} \mathsf{BL}_B(u) & \end{cases} \tag{9}$$

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 $A \to BC$. A similar reasoning as in the case of AF sets gives the following definition.

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

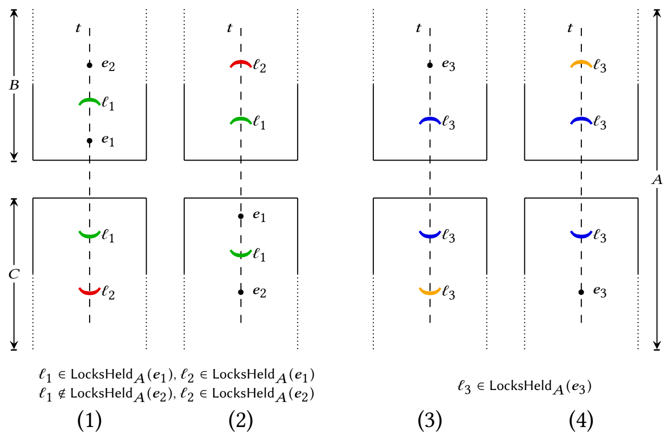
This completes the description of the HB-algorithm on compressed traces. Its correctness is proved in [30]. For a trace σ compressed as an SLP of size g, this algorithm runs in time $O(g(T+L)^2(L+TV))$ and uses space $O(g(T+L)(L+TV))$, where T, L and V denote the number of threads, locks and variables 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 (9).



t
e_2
ℓ_1
e_1
B
C
ℓ_1
ℓ_2
ℓ_2
ℓ_1
e_1
ℓ_1
e_2
e_3
ℓ_3
ℓ_3
ℓ_3
ℓ_3
ℓ_3
ℓ_3
e_3
A
$\ell_1 \in \mathsf{LocksHeld}_A(e_1)$, $\ell_2 \in \mathsf{LocksHeld}_A(e_1)$
$\ell_1 \notin \mathsf{LocksHeld}_A(e_2)$, $\ell_2 \in \mathsf{LocksHeld}_A(e_2)$
$\ell_3 \in \mathsf{LocksHeld}_A(e_3)$
(1)
(2)
(3)
(4)

Figure 5: Unmatched acquire/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)). '$\frown$ ℓ_i' represents $\mathtt{acq}(\ell_i)$, '$\smile$ ℓ_i' is $\mathtt{rel}(\ell_i)$.

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 occurring inside a chunk $[\![B]\!]$ are also violations of any other chunk that contains $[\![B]\!]$, and detecting "cross" violations is, therefore, the key challenge. In this section, we will outline these ideas in detail.

4.1 Cross Violations

Recall that, for a thread t and variable x, $\mathsf{LockSet}_\sigma(t,x)$ 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 show how to compute $\mathsf{LockSet}_A(t,x)$ 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 $\cap_{t\in\mathsf{Threads}A}\mathsf{LockSet}_A(t,x)=\emptyset$ then follows easily.

The base case for non-terminals with rule $A\to a$ is straightforward, and is presented in [30]. Now consider the inductive step for non-terminals having rules of the form $A\to BC$. To understand what $\mathsf{LockSet}_A(t,x)$ will be, it is useful to examine what $\mathsf{LocksHeld}_A(e)$ for an event e looks like. Consider a data access event $e\in[\![B]\!]$ performed by thread t. Clearly, $\mathsf{LocksHeld}_B(e)\subseteq\mathsf{LocksHeld}_A(e)$. But are they 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 $\mathsf{LocksHeld}_A(e)$ for every event $e\in B\lceil_t$. Lock ℓ_2 in Figure 5(1) illustrates this. Similarly, for an event $e\in[\![C]\!]$ performed by thread t, the set $\mathsf{LocksHeld}_A(e)$ 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 $e_{\mathtt{rel}(\ell)}$) and whose matching acquire is in B (event $e_{\mathtt{acq}(\ell)}$), does not affect the locks held by any event in B — for those events $e\in B\lceil_t$ which were after $e_{\mathtt{acq}(\ell)}$, ℓ was already in $\mathsf{LocksHeld}_B(e)$, while for the events e before $e_{\mathtt{acq}(\ell)}$, ℓ does not anyway protect e, and thus $\ell\notin\mathsf{LocksHeld}_B(e)$. 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

$$\mathsf{OpenAcq}_D(t,\ell)=|\{e=\langle t:\mathtt{acq}(\ell)\rangle\in D\mid \mathsf{match}(e)\notin[\![D]\!]\}| \quad (10)$$

and the number of release events as

$$\mathsf{OpenRel}_D(t,\ell)=|\{e=\langle t:\mathtt{rel}(\ell)\rangle\in D\mid \mathsf{match}(e)\notin[\![D]\!]\}| \quad (11)$$

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 $e\in B\lceil_t$ and $e'\in C\lceil_{t'}$ be read/write events performed by threads t,t'. Then,*

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

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

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

$$\begin{aligned}&\mathsf{LockSet}_A(t,x)\\&=\Big(\mathsf{LockSet}_B(t,x)\cup\{\ell\mid \mathsf{OpenRel}_C(t,\ell)>\mathsf{OpenAcq}_B(t,\ell)\}\Big)\\&\cap\Big(\mathsf{LockSet}_C(t,x)\cup\{\ell\mid \mathsf{OpenAcq}_B(t',\ell)>\mathsf{OpenRel}_C(t',\ell)\}\Big)\end{aligned} \quad (12)$$

The base case for computing $\mathsf{LockSet}_A(t,x)$ (see [30]).

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

$$\begin{array}{ll}\mathsf{OpenAcq}_W(1,\ell)=1 & \mathsf{OpenRel}_X(1,\ell)=1\\ \mathsf{OpenAcq}_X(2,\ell)=1 & \mathsf{OpenRel}_Z(2,\ell)=1\\ \mathsf{OpenAcq}_U(2,\ell)=1 & \mathsf{OpenRel}_V(2,\ell)=1.\end{array}$$

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 $\mathsf{OpenAcq}_U(1,\ell)=\mathsf{OpenRel}_U(1,\ell)=0$. This is essentially the insight we will explore in Section 4.2 to inductively define OpenAcq and OpenRel.

Let us now see how LockSet computation takes place. First, $\mathsf{LockSet}_W(1,x)=\{\Lambda,\Lambda_1\}$ since the only event of x in W is a read by thread 1. Also, $\mathsf{LockSet}_X(1,y)=\{\ell,\Lambda_1\}$ as e_3 is protected by the (unmatched) release e_4 in X. In chunk Y, $\mathsf{LockSet}_Y(2,x)=\{\Lambda,\Lambda_2\}$ and $\mathsf{LockSet}_Y(2,y)=\{\Lambda_2\}$; 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 $\mathsf{LockSet}_V(2,y)=(\mathsf{LockSet}_Y(2,y)\cup$

$\{\ell\}) \cap \top$ which evaluates to $\{\Lambda_2, \ell\}$ as expected. The universal set $\top$ is described in [30]. Again, the lock ℓ does not appear in $\mathsf{LockSet}_U(1, x)$ eventhough it is unmatched in X, because it gets matched with e_2 in W. This also follows from the inductive definition of $\mathsf{LockSet}_U(1, x)$.

Given Equation (12), 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 base case for non-terminals having rules of the form $A \to a$ is straightforward and can be found in [30]. 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 formalized below.

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

Notice the use of the max operator in Equation (13). If the quantity $\mathsf{OpenAcq}_B(t,\ell) - \mathsf{OpenRel}_C(t,\ell)$ is negative, then there are more unmatched $\mathtt{rel}(\ell)$-events in $C\!\upharpoonright_t$, which should be accounted for in $\mathsf{OpenRel}_A(t,\ell)$, instead of affecting the contribution of $[\![C]\!]$ towards the unmatched acquire events of $A\!\upharpoonright_t$.

Similar reasoning gives the inductive formulation for OpenRel

$$\begin{aligned}\mathsf{OpenRel}_A(t,\ell) = {} & \mathsf{OpenRel}_B(t,\ell)\\ & + \max\{0, \mathsf{OpenRel}_C(t,\ell) - \mathsf{OpenAcq}_B(t,\ell)\}\end{aligned} \tag{14}$$

This completes the description of our algorithm for computing locksets and checking violations of lockset discipline for compressed traces. 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 and Setup

Implementation. Our algorithms for detecting races on compressed traces, discussed in Section 3 and Section 4 have been implemented in our tool ZipTrack, which is publicly available at [4]. ZipTrack is written primarily in Java and analyzes 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 these SLPs to detect the presence of HB races and lockset discipline violations.

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.). Next, for production rules where the right hand side has both terminals and non-terminals and has long contiguous sequences (or substrings) of terminals, we introduce new production rules in the grammar, with fresh non-terminals corresponding to these long sequences. For example, for a rule of the form $A \to b_1 \cdots b_k C d_1 \cdots d_m$, where b_is and d_is are terminals, we will introduce two new 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 $A \to BCD$. This allows us to better exploit the vector-clock optimization.

Setup and Benchmarks. Our experiments were conducted on an 8-core 2.6GHz 64-bit Intel Xeon(R) Linux machine, with 30GB heap space. To compare against Happens-Before and LockSet based analysis on uncompressed traces, we use RAPID [3], which implements the standard Djit+ [43] vector clock algorithm, epoch optimizations like in FastTrack [20], the Goldilocks algorithm [12], and Eraser's lockset algorithm [49], as described in [43]. Our evaluation benchmarks (Column 1 in Table 1) are carefully chosen with the goal of being comprehensive, and have been primarily derived from [24]. The first set of small-sized (LOC ~ 50-300) benchmarks (account to pingpong) is derived from the IBM Contest benchmark suite [15]. The second set of medium sized (LOC ~ 3K) benchmarks (moldyn to raytracer) is derived from the Java Grande Forum benchmark suite [52]. The third set (derby to xalan) of benchmarks (LOC ~ 30K-500K) comes from the DaCaPo benchmark suite (version 9.12) [8] and large real world software including Apache FTPServer, W3C Jigsaw web server and Apache Derby. Columns 3, 4 and 5 in Table 1 report the number of threads, locks and variables in the traces generated from the corresponding programs in Column 1.

5.2 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 7 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

Table 1: Columns 1-5 describe the benchmarks and traces. Columns 6 and 7 describe the size of the compressed traces and the compression ratios achieved. Columns 8-10 describe the performance of various HB-race detection algorithms on uncompressed traces. Column 11-12 reports the performance of ZipTrack's HB race detection on compressed traces and the resulting speedup achieved. Column 13, 14 and 15 report the performance of respectively Eraser's lockset algorithm on uncompressed traces, ZipTrack's performance on compressed traces, and the resulting speedup achieved.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				Memory	Grammar	Compr.	HB (ms)					LockSet (ms)		
Program	Events	Threads	Locks	Loc.	Size	Ratio	Djit+	F.Track	Goldi.	Compr.	Speedup	Eraser	Compr.	Speedup
account	130	4	3	41	107	1.21	6	5	4	4	1	3	3	1
airline	137	4	0	44	132	1.04	8	4	5	6	0.67	2	1	2
array	47	3	2	30	47	1	5	4	4	3	1.33	3	1	3
boundedbuffer	337	2	2	63	194	1.74	8	8	2	18	0.11	3	2	1.5
bubblesort	4.2K	10	2	167	3.3K	1.29	14	13	12	92	0.13	2	4	0.5
bufwriter	11.8M	6	1	56	293	40238	20s	15.5s	36.3s	6	2600	1	4	0.25
critical	55	4	0	30	55	1	3	4	3	5	0.6	2	1	2
mergesort	3028	5	3	621	2795	1.08	5	7	5	13	0.38	3	6	0.5
pingpong	146	4	0	51	135	1.08	10	9	9	3	3	2	1	2
moldyn	164K	3	2	1197	88K	1.86	53	60	57	6	8.83	2	2	1
montecarlo	7.2M	3	3	876K	6.1M	1.18	317	271	302	87	3.11	300	1	300
raytracer	16.2K	3	8	3879	14.6K	1.11	58	32	32	315	0.1	25	133	0.19
derby	1.3M	4	1112	186K	735K	1.83	1006	1011	26s	592	1.7	848	<1	>1000
eclipse	90.6M	19	8300	11.2M	42.5M	2.13	34.6s	31.5s	3776s	17.4s	1.8	21737	1	21737
ftpserver	49K	11	301	5461	30K	2.13	49	44	91	23	1.9	34	1	34
jigsaw	3M	13	280	103K	908K	3.37	2432	2309	1888	195	9.7	12	4	3
lusearch	216M	7	118	5.2M	66.6K	3.25	1392	968	700	7	100	814	2	407
xalan	122M	6	2491	4.4M	71M	1.7	5183	3008	3709	109	27.6	2779	1	2779

> 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 8, 9 and 10 in Table 1 represent the time taken to detect the presence of an HB race by respectively, Djit+, FastTrack and Goldilocks. Column 11 denotes the time taken by our HB race detection algorithm for analyzing the traces compressed as SLPs and Column 12 reports the speedup achieved over the best of the three values in Columns 8, 9 and 10.

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 – 8x, 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 very early 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 100x. The Fast-Track vector clock algorithm [20] 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 [20]). 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 13 and 14 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, 000x, 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; compression can be leveraged not only for smaller storage spaces, but also for a more efficient race detection 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.

ACKNOWLEDGMENTS

We gratefully acknowledge National Science Foundation for supporting Umang Mathur (grant NSF CSR 1422798) and Mahesh Viswanathan (NSF CPS 1329991).

REFERENCES

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

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

[3] 2018. RAPID: Dynamic Analysis for Concurrent Programs. https://github.com/umangm/rapid. Accessed: July 30, 2018.

[4] 2018. ZipTrack: Race Detection on Compressed Traces. https://github.com/umangm/ziptrack. Accessed: July 30, 2018.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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*.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2018. Data Race Detection on Compressed Traces. *CoRR* abs/1807.08427 (2018). http://arxiv.org/abs/1807.08427

[31] 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.

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

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

[34] 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.

[35] 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.

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

[37] 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).

[38] 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.

[39] 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.

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

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

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

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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.

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