Data Race Detection on Compressed Traces

Dileep Kini
Akuna Capital
USA

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.

ACM Reference Format:

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].

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 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.

¹We 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.
Figure 1: A simple concurrent program in Java

```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 (expected) = " + ITERS*2);
        System.out.println("y (actual) = " + y);
    }
}
```

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 compression 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 re-orderings 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 \( t : o \), where \( t \) is the thread performing the operation \( o \). Operations can be one of the following: forking of a new child thread (fork \( (t) \)), joining of a child thread \( (join(t)) \), acquiring and releasing a lock \( (acq(t) \) and \( rel(t)) \); and, reading and writing to a variable \( (r(x) \) and \( w(x)) \). 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 \( e = (t : rel(l)) \), we can associate a unique acquire event \( e^r = (t : acq(l)) \), which is the last \( acq(l) \)-event in thread \( t \) before \( e \) that is not matched with any \( rel(l) \)-event in thread \( t \) before \( e \). This \( acq(l) \)-event \( e^r \) is said to be the matching acquire of \( e \), and is denoted by \( match(e) \). Similarly, for an acquire event \( e^r \) such that \( e^r = match(e) \), we will say that \( e \) is the matching release of \( e^r \), and we will also denote this by \( match(e^r) \). For a trace \( \sigma, \sigma^r \) will denote the subsequence of events performed by thread \( t \).

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

\[
Threads(\sigma) = \{ t \mid \exists e \in \sigma. e = (t : o) \text{ or } e = (t' : \text{fork}(t)) \text{ or } e = (t' : \text{join}(t)) \}.
\]

Formally, each event in a trace is assumed to have a unique event id. Thus, two occurrences 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.
Data Race Detection on Compressed Traces

Fig. 2: Example trace $\sigma_1$ and its SLP representation

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

For a non-empty subset of events $S$, we will denote by $\text{Last}_\sigma(S)$ the (unique) event $e \in S$, that is latest in $\sigma$ amongst the events in $S$. Similarly, $\text{First}_\sigma(S)$ is the event $e \in S$ that is earliest in $\sigma$ amongst the events in $S$. When $\sigma$ is empty, we say both $\text{First}_\sigma(S)$ and $\text{Last}_\sigma(S)$ are undefined.

Example 2.1. We illustrate the above definitions on the example trace $\sigma_1$ 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 $e_1$ to denote the $i$th event in $\sigma_1$. Let $\sigma_1 = \text{REvents}_2(2, x) = \{e_3, e_11\}$ and $\sigma_2 = \text{WEvents}_2(y) = \{e_5, e_{10}, e_{13}, e_{16}\}$. The set $\text{Rd}(\sigma_1) = \{(1, x), (2, x)\}$ while $\text{Wr}(\sigma_1) = \{x, y\}$. Finally, $\text{Last}_{\sigma_1}(\{1\}) = e_{11}$, and $\text{First}_{\sigma_1}(\{2\}) = e_3$. Orders on Traces. Let us fix a trace $\sigma$. If an event $e_1$ appears earlier in the sequence $\sigma$ than $e_2$, then we say $e_1$ is trace ordered before $e_2$ and denote it as $e_1 <_{T} e_2$. We say $e_1$ is thread ordered before $e_2$, denoted by $e_1 <_{T} e_2$, if $e_1$ and $e_2$ are events performed by the same thread and $e_1 <_{T} e_2$. 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 $\sigma$ said to happen before event $e'$ in $\sigma$, denoted $e <_{HB} e'$, 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 $(e_i, e_{i+1}) (i < n)$, $e_i <_{T} e_{i+1}$ and one of the following conditions hold:

1. $e_i <_{T} e_{i+1}$,
2. $e_i = (t : \text{rel}(t))$ and $e_{i+1} = (t' : \text{acq}(t'))$ for some $t, t', e$,
3. $e_i = (t : \text{fork}(t'))$ and $e_{i+1} = (t' : o)$ for some $t, t', o$, or
4. $e_i = (t' : o)$ and $e_{i+1} = (t : \text{join}(t'))$ for some $t, t', o$.

For any $P \in \{T, TO, HB\}$, $\leq_P$ refers to the partial relation $<_{P}$

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

We now define races identified by the happens-before relation. A pair of events $e_1 = (t_1 : a_1(x))$ and $e_2 = (t_2 : a_2(x))$ (for some variable $x$) is said to be conflicting, denoted $e_1 \times e_2$, if $t_1 \neq t_2$ and at least one out of $a_1$ and $a_2$ is $w$. A trace $\sigma$ 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 $e_1 \times e_2$ and $e_1||e_2$.

Example 2.3. We illustrate the happens before relation through the trace $\sigma_1$ in Figure 2. $e_1 \parallel_{HB} e_3$ because $e_2$ happens before every event in thread $2$ since it forks thread $2$. Similarly, we can conclude that $e_{13} \parallel_{HB} e_{16}$ because the join event $e_{13}$ is after every event in thread $2$. Another interesting pair is $e_5 \parallel_{HB} e_{10}$. This is because $e_4, e_5, e_6$, and $e_8, e_9$ are critical sections over the same lock $l$, and thus, $e_5$ happens before $e_8$. Therefore, $e_5 \parallel_{TO} e_8 \parallel_{HB} e_5 \parallel_{TO} e_{10}$. It is useful to pay attention to a couple of concurrent pairs of events. Events $e_2$ and $e_3$ 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 $O(T + L + T \log n)$ 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 $\text{After}_\sigma$ defined as follows: $\text{After}_\sigma(e) = \{t \in \text{Threads}(\sigma) | \exists e' = (t : o), e <_{HB} e'\}$

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

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

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

and for every variable $x$ in $\sigma'$, the set $\text{GLS}_{\sigma'}^W(x) = \text{After}_{\sigma'}(\text{Last}_{\sigma'}(\text{WEvents}_{\sigma'}(x)))$

where, $\text{After}_{\sigma'}$(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. $e = (t : w(x))$ and either $t \not\in \text{GLS}_{\sigma'}^W(x)$ or $t \not\in \text{GLS}_{\sigma'}^R(t', x)$ for some thread $t' \in \text{Threads}(\sigma')$
2. $e = (t : r(x))$ and $t \not\in \text{GLS}_{\sigma'}^W(x)$.

where $\sigma'$ is the prefix until the event $e$. This algorithm runs in time $O(nL + TV)$ 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
We conclude this description by highlighting the importance of 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 \( D = \{ \Lambda \} \cup \{ \Lambda_1 \mid t \text{ is a thread} \} \) are locks used by the program. The elements of the set \( D \) 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 \( e = (t : a(x)) \) (where \( a \) is either \( r \) or \( w \)) in trace \( \sigma \), \( \text{LocksHeld}_\sigma(e) \) is the set of locks held by \( t \) when \( e \) is performed. Using this, for an event \( e = (t : a(x)) \) we define \( \text{LockSet}_\sigma(e) \) to be

\[
\text{LockSet}_\sigma(e) = \begin{cases} 
\{ \Lambda, \Lambda_1 \} \cup \text{LocksHeld}_\sigma(e) & \text{if } a = r \\
\{ \Lambda_1 \} \cup \text{LocksHeld}_\sigma(e) & \text{if } a = w
\end{cases}
\]

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

\[
\text{LockSet}_\sigma(t, x) = \bigcap_{e \in \text{Access}_\sigma(t, x)} \text{LockSet}_\sigma(e).
\]

As per convention, when \( \text{Access}_\sigma(t, x) = \emptyset \) (i.e., thread \( t \) never accesses the variable \( x \)), the right hand side of the above equation is assumed to be \( \text{Locks}(\sigma) \cup D \). A few observations about these definitions are in order. First, \( \text{LockSet}_\sigma(t, x) \) is always non-empty because \( \Lambda_1 \in \text{LockSet}_\sigma(t, x) \). Second, if all events in \( \text{Access}_\sigma(t, x) \) are read events, then \( \Lambda \in \text{LockSet}_\sigma(t, x) \). The lockset discipline is said to be violated in trace \( \sigma \), if for some variable \( x \),

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

Note that the Eraser algorithm crucially depends upon the accurate computation of \( \text{LocksHeld}_\sigma(e) \). For this, we need to record, for each thread \( t \) and lock \( \ell \), the number of times \( \ell \) 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 \( D \) that were introduced. Take \( \text{Locks} = \bigcap_{t \in \text{Threads}(\sigma)} \text{LockSet}_\sigma(t, x) \).

If a variable \( x \) is only accessed by a single thread \( t_1 \), then \( \text{Locks} \) is non-empty because it contains \( \Lambda_1 \). And if a variable \( x \) is only read and never written to, then \( \text{Locks} \) is again non-empty because it contains \( \Lambda \). The Eraser algorithm [46] checks for violation of the lockset principle by maintaining the lockset for each thread-variable pair. It runs in time \( O(nL + \log r) \) 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 \( \sigma_2 \) in Figure 3. The relevant locksets are as follows.

- \( \text{LockSet}_{\sigma_2}(1, x) = \{\Lambda, \Lambda_1\} \)
- \( \text{LockSet}_{\sigma_2}(2, x) = \{\Lambda, \Lambda_2\} \)
- \( \text{LockSet}_{\sigma_2}(1, y) = \{\Lambda_1, \ell\} \)
- \( \text{LockSet}_{\sigma_2}(2, y) = \{\Lambda_2, \ell\} \)
- \( \text{LockSet}_{\sigma_2}(1, z) = \{\Lambda_1\} \)
- \( \text{LockSet}_{\sigma_2}(2, z) = \{\Lambda, \Lambda_1, \Lambda_2, \ell\} \)

Observe that \( \text{LockSet}_{\sigma_2}(2, z) \) is the set of all locks because thread 2 does not access \( z \). The trace \( \sigma_2 \) 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 \( \ell \), and variable \( z \) is local to thread 1. Trace \( \sigma_2 \) also contains no HB-race.

For trace \( \sigma_1 \) from Figure 2,

- \( \text{LockSet}_{\sigma_1}(1, x) = \{\Lambda_1\} \)
- \( \text{LockSet}_{\sigma_1}(1, y) = \{\Lambda_1\} \)
- \( \text{LockSet}_{\sigma_1}(2, x) = \{\Lambda, \Lambda_2\} \)
- \( \text{LockSet}_{\sigma_1}(2, y) = \{\Lambda_2, \ell\} \)

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 a 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 \( A \to BC \), \( A < B \) and \( A < C \). 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, \( \text{Locks}(A) \) will mean \( \text{Locks}([A]) \).

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 \( \{A_{i,[i]} \mid 1 \leq i \leq n\} \cup \{A_{i,[i]} \mid 1 \leq i \leq n\} \) with start symbol \( A_{1,[1]} \). Intuitively, \( A_{i,[i]} \) represents the string \( e_i \), while \( A_{i,[i]} \) represents the prefix of length \( i \). This is accomplished by the rules \( \rightarrow A_{i,[i]} \to e_i \) and \( A_{i,[i]} \to A_{i,1,[i-1]}A_{i,i} \) for each \( 1 \leq i \leq n \). Second, the SLP representation of a string \( \sigma \) may be exponentially smaller than \( \sigma \) itself. For example, take \( \sigma = a^n \). An \( O(n) \) SLP
representation for \( \sigma \) is as follows: \( N = \{ A_i \mid 0 \leq i \leq n \} \) with rules \( A_0 \rightarrow a \) and \( A_{i+1} \rightarrow A_i A_i \). One can inductively observe that \( \|A_0\| = a^2 \), and so \( \|A_n\| = \sigma \).

Example 2.5. Figure 2 describes an SLP representation of trace \( \sigma_1 \). 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 \( i \)th event of \( \sigma_1 \). Chunk \( E \) represents \( e_1, e_2, F \) represents \( e_3, e_4, e_5, e_6 \) and \( e_{11} \). \( D \) 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, Threads\((E) = \{1, 2\} \).

Similarly, the SLP for \( \sigma_2 \) 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 occurrence 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 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 \( A \rightarrow a \), where \( a \) is an event, Race\(?(A) \) is clearly false, because \([A]\), in this case, has only one event.

![Figure 4: Illustrating the various scenarios that establish \( e_j \).](image)

Let us now consider the case when \( A \) has the rule \( A \rightarrow BC \), 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 Race\(?(B) \) and 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 \( e \prec e' \). 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 \( e_i \prec e_{i+1} \). Also, \( e \in e_1 \in [B] \), and \( e_n = e' \). 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 \( e = e_1 \prec e_j \) and \( e_j \prec e_n = e' \). 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 \([B] \cup [C] \). 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 \( \sigma \), and event \( e \in \sigma \), recall the function \( \text{After}_\sigma \):

\[
\text{After}_\sigma(e) = \{ t \in \text{Threads}(\sigma) \mid 3e' = (t : o), e \prec e' \}
\]

\[
\cup \{ t \in \text{Threads}(\sigma) \mid 3e' = (t' : \text{fork}(t)), e \prec e' \}
\]

\[
\cup \{ t \in \text{Locks}(\sigma) \mid 3e' = (t : \text{rel}(t)), e \prec e' \}
\]

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

\[
\text{Before}_\sigma(e) = \{ t \in \text{Threads}(\sigma) \mid 3e' = (t : o), e \preceq e' \}
\]

\[
\cup \{ t \in \text{Threads}(\sigma) \mid 3e' = (t' : \text{join}(t)), e \preceq e' \}
\]

\[
\cup \{ t \in \text{Locks}(\sigma) \mid 3e' = (t : \text{acq}(t)), e \preceq e' \}
\]
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 $e' \in [C]$. $e \leq_{HB} e'$ iff $\text{After}_B(e) \cap \text{Before}_C(e') \neq \emptyset$.

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$_B^e$ and first$_C^{e'}$, where

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

and

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

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 $\text{ALRd}_t$, $\text{ALWr}_t$, $\text{BFRd}_t$ and $\text{BFWr}_t$ Formally,

$$
\begin{align*}
\text{ALRd}_t(t,x) &= \text{After}_t(\text{REvents}_t(t,x)) \\
\text{ALWr}_t(x) &= \text{After}_t(\text{WEvents}_t(x)) \\
\text{BFRd}_t(t,x) &= \text{Before}_t(\text{REvents}_t(t,x)) \\
\text{BFWr}_t(x) &= \text{Before}_t(\text{WEvents}_t(x))
\end{align*}
$$

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

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

$$
\begin{align*}
\text{Race}?(A) &= \text{Race}?(B) \lor \text{Race}?(C) \\
\forall x \in \text{Wr}(B) \lor \text{Wr}(C) &\quad \text{ALWr}_t(x) \land \text{BFWr}_t(x) = \emptyset \\
\forall x \in \text{Wr}(B), (t,x) \in BFRd_t &\quad \text{ALRd}_t(t,x) \land \text{BFRd}_t(t,x) \land \text{BFWr}_t(x) = \emptyset \\
\forall (t,x) \in \text{BFRd}(B), x \in \text{Wr}(C) &\quad \text{ALRd}_t(t,x) \land \text{BFWr}_t(x) = \emptyset
\end{align*}
$$

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

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

**Example 3.4.** Let us illustrate the ideas presented in this section through some examples. We will consider traces $\sigma_1$ and its SLP in Figure 2, and $\sigma_2$ with its SLP in Figure 3.

We begin by giving examples of Before and After sets.

- $\text{After}_B(e_1) = \{1, 2\}$, $\text{After}_B(e_2) = \{1, 2, \ell\}$
- $\text{Before}_B(e_1) = \{1, 2\}$, $\text{Before}_B(e_2) = \{1, 2, \ell\}$
- $\text{After}_B(e_3) = \{1\}$, $\text{After}_B(e_4) = \{1, 2, \ell\}$
- $\text{Before}_B(e_3) = \{1\}$, $\text{Before}_B(e_4) = \{1, \ell\}$

Let us highlight the significant aspects of these examples. $2 \in \text{After}_B(e_1)$ because of $e_2 = \langle 1 : \text{fork}(2) \rangle$ and $\ell \in \text{After}_B(e_2)$ because of event $e_3 = \langle 2 : \text{rel}(\ell) \rangle$. On the other hand, $\ell \notin \text{After}_B(e_3)$ because there is no $\text{rel}(\ell)$ event in chunk $[W]$. But when considering the chunk $[U]$ of $\sigma_2$, we have $\ell \notin \text{After}_B(e_4)$ because of the event $e_4 = \langle 1 : \text{rel}(\ell) \rangle$. Next, $2 \in \text{Before}_B(e_1)$ because of join event $e_{13}$, and $\ell \notin \text{Before}_B(e_4)$ because of acquire event $e_{12}$.

In trace $\sigma_2$, $\ell \notin \text{Before}_B(e_3)$ 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 $\text{Race}?(M) = \text{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 $\text{ALWr}_F(x) = \{1, 2\}$ and $\text{BFRd}_F(2, x) = \{2\}$, both of which have the thread $2$ in common, and thus the intersection $\text{ALWr}_F(x) \cap \text{BFRd}_F(x)$ is non-empty. In fact, what this reasoning demonstrates is that there is no race between a $w(x)$-event in $E$ and a $r(x)$-event in $F$. Similarly, the absence of a race between $e_{15}$ and $e_{16}$ can be seen because $\text{ALWr}_F(y) \cap \text{BFRd}_F(1, y) = \emptyset$. To reason about the events $e_{15}$ and $e_{16}$, observe that $\text{ALWr}_F(y) = \{2, \ell\}$ and $\text{BFWr}_F(y) = \{1, \ell\}$, both of which have the $\ell$ in common. Thus, we can conclude there is no race between a pair of $w(y)$-events crossing the chunk $FD$.

Our reasoning also reveals the existence of HB-concurrent events. For example, $\text{ALRd}_D(2, x) = \{2, \ell\}$, and $\text{BFRd}_D(1, x) = \{1\}$. Since these sets are disjoint, it reveals that there are a pair of $r(x)$-events (namely, $e_3$ and $e_5$) that are HB-concurrent; it is not an HB-race because these events are not conflicting (none of $e_3$ and $e_5$ is a write event). The race between $e_{10}$ and $e_{14}$ can be seen as follows. $\text{ALWr}_A(y) = \{1\}$, and $\text{BFWr}_B(y) = \{2, \ell\}$. 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 $\text{ALRd}_t$, $\text{ALWr}_t$, $\text{BFRd}_t$, and $\text{BFWr}_t$ (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 $\text{ALRd}_t$ and $\text{BFRd}_t$. The computation of the sets $\text{ALWr}_t$ and $\text{BFWr}_t$ is similar and is presented completely in the Appendix.

The base case for non-terminals with rule $A \rightarrow a$, 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 $A \rightarrow BC$.

First consider the case of $\text{ALRd}_t(t,x)$, which is equal to the after set $\text{After}_A(e)$, where $e$ is the last event amongst the read events $\text{REvents}_t(t,x)$. If there are no read events in $A$ by thread $t$ on variable $x$ (i.e., $\text{REvents}_t(t,x) = \emptyset$), we will have $\text{ALRd}_t(t,x) = \emptyset$. 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 $\text{ALRd}_t(t,x) = \text{After}_B(e)$. Observe that since $\exists e' \in [B]$, $e' \leq_{HB} e$, we have $\text{After}_A(e) = \text{After}_B(e)$. Thus, in this case, $\text{ALRd}_t(t,x) = \text{ALRd}_C(t,x)$. The interesting case is when $\text{REvents}_t(t,x)$ is empty and $\text{REvents}_t(t,x) = \emptyset$, i.e., the last read event $e$ belongs to the
chunk $B$. Since $\{e' \in [B] \mid e \leq_{HB} e'\} \subseteq \{e' \in [A] \mid e \leq_{HB} e'\}$, we have $\text{After}_{B}(e) \subseteq \text{After}_{A}(e)$. Consider $e' \in [C]$ such that $e \leq_{HB} e'$. 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 $e \leq_{HB} e_1$, $e_2 \leq_{HB} e'$, 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 $\text{After}_{B}(e)$, and $e'$ is $\text{HB}$-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

$$AF_D(t) = \text{After}_D(\text{First}_D(\text{ThEvents}^\text{join}_D(t)))$$

(3)

where $\text{ThEvents}^\text{join}_D(t)$ is the set $\{e \in D \mid e = (t: o) \text{ or } e = (t': \text{fork}(t))\}$. Similarly, for a lock $\ell$, define

$$AF_D(t) = \text{After}_D(\text{First}_D(\text{AcqEvents}_D(\ell)))$$

(4)

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

We now formalize our intuitions in the following.

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

$$\text{After}_A(e) = \text{After}_B(e) \cup \bigcup_{u \in \text{After}_A(e)} AF_C(u)$$

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

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

(5)

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

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

$$AF_A(u) = AF_B(u) \cup \bigcup_{u' \in \{u \cup AF_B(u)\}} AF_C(u')$$

We conclude the description of the HB-algorithm for compressed traces, by outlining the inductive definition of the set $\text{BFrd}_A(t, x)$ for thread $t$ and variable $x$. As before, the first event $e \in [A]$ of the kind $(t : r(x))$ can either belong to $[B]$ or to $[C]$. When $e \in \text{REvents}_A(t, x)$, we have $\text{Before}_A(e) = \text{Before}_B(e)$, since $e' | e' \leq_{HB} e = \{e' | e' \leq_{HB} e\}$. On the other hand, if $e \in [C]$, in a manner similar to the case for $\text{After}_B(e)$, we need to “compose” $\text{Before}_C(e)$ 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 $t$,

$$\text{BLD}(t) = \text{Before}_D(\text{Last}_D(\text{ThEvents}^\text{fork}_D(t)))$$

(6)

where $\text{ThEvents}^\text{fork}_D(t) = \{e \in D \mid e = (t : o) \text{ or } e = (t' : \text{fork}(t))\}$. For a lock $\ell$,

$$\text{BLD}(\ell) = \text{Before}_D(\text{Last}_D(\text{RelEvents}_D(\ell)))$$

(7)

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

As in Lemma 3.5, the sets $\text{BLD}(u)$ can be used to compute $\text{Before}_C(e)$ for any event $e \in [C]$.

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

$$\text{Before}_A(e) = \text{Before}_C(e) \cup \bigcup_{u \in \text{Before}_C(e)} \text{BLD}(u)$$

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

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

(8)

To complete the algorithm, we need to give the inductive definition of $\text{BLD}(u)$ for thread/lock $u$. Again the interesting case is the inductive case of a non-terminal $A$ with rule $A \rightarrow BC$. 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.

$$\text{BLD}(u) = \text{BLD}(u) \cup \bigcup_{u' \in \{u \cup \text{BLD}(u)\}} \text{BLD}(u')$$

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

**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 $\sigma_1$ and its SLP grammar in Figure 2. Let us consider the computation of $\text{ALW}_C(x)$. Observe that the last $w(x)$-event in $C$ is $e_1$. Further,

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

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

Next, consider the computation $\text{BFrd}_B(y)$. Notice that the first $w(y)$-event in $B$ is $e_3$, which is in the chunk $F$. This immediately gives $\text{BFrd}_B(y) = \text{BFrd}_B(y) = \{2, \ell\}$ using a characterization similar to Equation (8).
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 \( \ell \) which was released by \( t \) in \([C]\) (at event \( e_{\ell}(\ell) \)) and whose matching acquire is in \([B]\) (event \( e_{acq}(\ell) \)) does not affect the locks held by any event in \([B] \) for those events \( e \in [B]_{t}\) which were after \( e_{acq}(\ell) \). \( \ell \) was already in \( \text{LocksHeld}_B(e) \), while for the events \( e \) before \( e_{acq}(\ell) \), \( \ell \) does not anyway protect \( e \), and thus \( \ell \notin \text{LocksHeld}_B(e) \). This is illustrated through lock \( \ell_1 \) 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 \( \ell \) in thread \( t \) as

\[
\text{OpenAcq}_D(t, \ell) = |\{ e \in [D] | e_{acq}(\ell) \in E, \text{ and } e \notin [D] \} | \tag{9}
\]

and the number of release events as

\[
\text{OpenRel}_D(t, \ell) = |\{ e \in [D] | e_{rel}(\ell) \in E, \text{ and } e \notin [D] \} | \tag{10}
\]

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 \( A \rightarrow BC \). Let \( e \in [B]_{t} \) and \( e' \in [C]_{t} \) be read/write events performed by threads \( t, t' \). Then,

\[
\text{LocksHeld}_A(e) = \text{LocksHeld}_B(e) \cap \text{LocksHeld}_C(e')
\]

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

\[
\text{LockSet}_A(t, x) = (\text{LockSet}_B(t, x) \cup \{ \ell | \text{OpenRel}_C(t, \ell) > \text{OpenAcq}_B(t, \ell) \})
\]

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

**Example 4.2.** Consider the SLP for \( \sigma_2 \) from Figure 3. \( \text{OpenAcq} \) and \( \text{OpenRel} \) for the different non-terminals is given below:

\[
\begin{align*}
\text{OpenAcq}_W(1, \ell) &= 1, \text{OpenRel}_X(1, \ell) = 1 \\
\text{OpenAcq}_X(2, \ell) &= 1, \text{OpenRel}_Z(2, \ell) = 1 \\
\text{OpenAcq}_Z(2, \ell) &= 1, \text{OpenRel}_V(2, \ell) = 1.
\end{align*}
\]

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, \( \text{LockSet}_W(1, x) = \{A, A_1\} \) since the only event of \( x \) in \( W \) is a read

---

**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)]. '\( \sim \) \( \ell \)' represents \( acq(\ell), '\sim \) \( \ell \)' is \( rel(\ell) \).

**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 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.

**4.1 Cross violations**

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

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 \( \bigcap_{t \in \text{Threads}_A} \text{LockSet}_A(t, x) = \emptyset \) then follows easily.

The base case for non-terminals with rule \( A \rightarrow \alpha \) is straightforward, and is presented in the Appendix. Now consider the inductive step for non-terminals having rules of the form \( A \rightarrow BC \). 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 \( \ell \) 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 \( \ell \) 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, \( \ell \) must be included in the set \( \text{LocksHeld}_A(e) \) for every event \( e \in [B]_{t} \). Lock \( l_2 \) in Figure 5(1) illustrates this. Similarly, for an event \( e \in [C] \) performed by thread \( t \), the set \( \text{LocksHeld}_A(e) \)
by thread 1. Also, LockSet_Y(1, y) = {ℓ, A_1} as e_3 is protected by the (unmatched) release e_4 in X. In chunk Y, LockSet_Y(2, x) = {A_2, A_2} and LockSet_Y(2, x) = {A_3}; interestingly, the locksets for Y do not reveal that both e_2 and e_4 are enclosed within the critical section of lock ℓ. On the other hand, using the inductive formulation discussed above, we can infer that LockSet_Y(2, y) = (LockSet_Y(2, y) ∪ {ℓ}) ∩ τ which evaluates to {A_2, ℓ} as expected. The universal set τ is described in Definition ?? (See the Appendix). Again, the ℓ does not appear in LockSet_Y(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 LockSet_Y(1, x) because OpenRel_Y(1, ℓ) ⊄ OpenAcq_W(1, ℓ).

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 → 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 → 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

\[ \text{OpenAcq}_A(t, ℓ) = \text{OpenAcq}_C(t, ℓ) + \max(0, \text{OpenAcq}_B(t, ℓ) - \text{OpenRel}_C(t, ℓ)) \]  

(12)

Notice the use of the max operator in Equation (12). If the quantity OpenAcq_B(t, ℓ) − OpenRel_C(t, ℓ) is negative, there are more unmatched release events of ℓ in C[τ] 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[τ].

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

\[ \text{OpenRel}_A(t, ℓ) = \text{OpenRel}_B(t, ℓ) + \max(0, \text{OpenRel}_C(t, ℓ) - \text{OpenAcq}_B(t, ℓ)) \]  

(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 first 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 → a_1a_2⋯a_k, where each of a_1, a_2⋯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 → b_1⋯b_kCd_1⋯d_m, where b_is and d_is are terminals, we will introduce two non-terminals B and D, with production rules B → b_1⋯b_k and D → d_1⋯d_k, and replace the production rule of A by A → 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].
Table 1: Benchmarks: Trace size, compressed grammar size and compression ratios.

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.

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 bufferwriter (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 bufferwriter, 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 speedup 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 bufferwriter 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 – 4x, 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 (e1, e2) 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 up to 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; 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 SLFs. 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
This material is based upon work supported by the National Science Foundation under Grant No. nmmmnn and Grant No. mmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES


