

© 2014 Yang Xu

ADAPTATION OF GRAMMAR BASED COMPRESSION IN
DETECTING ATOMICITY VIOLATIONS

BY

YANG XU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Associate Professor Mahesh Viswanathan

ABSTRACT

Atomicity violation is a common kind of concurrency bug in real world projects. **PENELOPE** is a framework that can detect atomicity violations in a single observed trace, without explicitly examining all traces that result from every possible interleaving. This thesis proposes a way to improve performance of **PENELOPE**'s prediction stage by performing computation directly on grammar compressed execution trace file, leading to a running time linear in the length of compressed file and size of grammar.

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	MODEL OF MULTI-THREADED PROGRAMS	3
CHAPTER 3	ALGORITHM OF PENELOPE	6
3.1	Formal Definition of Atomicity	6
3.2	Three-access Atomicity Violation	7
CHAPTER 4	STRAIGHT LINE GRAMMAR	11
CHAPTER 5	ALGORITHM ON GRAMMAR COMPRESSED	
TRACE		13
5.1	Lock Value Difference and Last Acquisition	13
5.2	Computing Indices for Events of Interest	16
5.3	Getting Lock Sets and Acquisition Histories for All Events . .	19
CHAPTER 6	SELECTION OF STRAIGHT LINE GRAMMAR	21
CHAPTER 7	EXPERIMENTS	24
CHAPTER 8	CONCLUSION	26
REFERENCES		27

CHAPTER 1

INTRODUCTION

As hardware engineers stopped pursuing higher CPU frequency for the sake of power consumption, multi-core architecture became the solution to improving CPU performance. The introduction of multi-core architecture has greatly changed the way programs are written. Programers are encouraged to write multi-threaded code to take advantage of multi-core hardware as multi-threaded libraries are becoming increasingly popular.

Despite the advantage of being able to exploit parallelism in computation, multi-threaded programs are difficult to develop. Debugging or even detecting concurrent bugs can be challenging. Unlike traditional sequential programming where bugs can be deterministically reproduced, concurrency bugs often occur only with a particular interleaving of instructions, making it harder to reproduce and harder for programmers to reason about. Also because of *state space explosion* – the exponential growth of number of program states with respect to the number of concurrent components, it is difficult to build debugging tools tracking states of multi-threaded programs.

Atomicity violations are concurrency bugs caused by nonexclusive access of shared resources. In [1], a study to real world code from open source projects, including MySQL, Apache, Mozilla and OpenOffice, shows that more than 2/3 or real world concurrency bugs are atomicity violations.

PENELOPE, introduced in [2], tests specifically for *three-access atomicity violations* among threads interacting using nested locking. It works by first having an arbitrary run involving multiple threads of program being verified, getting an *abstracted* trace from that execution where events like thread creation, thread termination, reading shared variable, writing shared variable etc. are recorded but details like the exact value being written to the shared variable are omitted. On this level of abstraction, it is possible to capture all five patterns of atomicity violation and algorithmically generate all potential violating schedules which will later be verified.

Our work is inspired by the observation that abstracted traces used by PENELOPE have very few unique events compared to its length thus can be efficiently compressed. It will be an improvement if we can verify atomicity safety in time with respect to the length of compressed trace, rather than the length of original trace.

Trace files are usually stored in some compressed form in practice. Lossless compression is often done by exploiting repeated patterns in input string, replacing multiple occurrences of such patterns with more space efficient symbols and a derivation rule that can be used to reconstruct input string. In [3], the category of lossless data compression via the construction of a context free grammar specifically for the input string, is formalized as *grammar based codes*. Context free grammars that generate exactly one string are called *straight line grammars* (SLG). The popular LZ-family compression algorithms are good representatives of grammar based codes. Grammar based codes include, in addition to classical LZ-family compression algorithms, more recent algorithms such as **Sequitur** [4] and **Re-Pair** [5].

For some operations, the same input substring always has the same effect to system's change of state, in which case storing intermediate computation results of repeated substrings for future reuse is an optimization. Grammar based codes can help discover structural hierarchy in input string and thus repeated substrings [6]. In particular, the fact that a substring corresponds to a nonterminal node in grammar suggests that the substring has multiple appearances. The main contribution of this thesis is exploiting such operations in the process of atomicity violation detection and reuse intermediate computation results on repeated input substrings suggested by the grammar compressing input string.

In this thesis, we identify the operations that can be performed efficiently directly on grammar compressed abstract trace files without explicit decompression, and show both asymptotic and practical improvement.

CHAPTER 2

MODEL OF MULTI-THREADED PROGRAMS

The system studied is a simplified model of multi-threaded programs where threads only communicate by using locks and lock accesses are nested. A thread is modeled as *control flow graphs* (CFG) with stacks. Vertices of CFGs are *basic blocks*, which is code not containing flow control statements, such as **if**, **for**, **while** keywords in C-like languages. Edges of CFG are flow control statements. Recursions and mutual recursions are possible with calling stacks. Every fixed size program's CFG is finite.

Formally, a single threaded program P_i can be described as a *pushdown system* (PDS). A PDS is a five-tuple: $P_i = (L_i, A_i, \Gamma_i, c_0^i, T_i)$. L_i is the set of locations, corresponds to points between any two instructions in basic blocks, representing states of program execution; A_i is the set of actions, such as variable assignment and function invocation etc. (notice lock acquisition and release are not included); Γ_i is the stack alphabet; $T_i : L_i \times \Gamma_i \times A_i \times L_i \times \Gamma_i^*$ is the set of transition rules. A configuration $c^i \in L_i \times \Gamma_i^*$ is a tuple (o, w) where o is a location and w is stack content. c_0^i is the initial configuration of P_i . $\xrightarrow{a}$ is a relation between configurations, and $(o, wv) \xrightarrow{a} (o', w'v)$ if $(o, w, a, o', w') \in T_i$ for any $v \in \Gamma_i^*$.

Let $\mathcal{L}$ be the set of locks and $\mathcal{T}$ be the set of threads. For program P with $|\mathcal{T}|$ threads, $t_1, t_2, \dots, t_{|\mathcal{T}|} \in \mathcal{T}$, a lock $l \in \mathcal{L}$ is a globally shared variable that can have value $\eta(l) \in \{1, 2, \dots, |\mathcal{T}|, \perp\}$. $\eta(l) = i$ when lock l is acquired by thread t_i and $\eta(l) = \perp$ when it is free. A lock with value other than $\perp$ can not be acquired. A lock with value i can only be released by thread t_i and a lock with value $\perp$ can not be released. At program initialization, all locks are free. Denote acquiring lock l as $\mathbf{a}(l)$ and releasing lock l as $\mathbf{r}(l)$.

A multi-threaded program P with $|\mathcal{T}|$ threads and $|\mathcal{L}|$ locks can be represented as a tuple $P = (P_1, P_2, \dots, P_{|\mathcal{T}|}, \eta(l_1), \eta(l_2), \dots, \eta(l_{|\mathcal{L}|}))$ whose initial state, or global initial state $c_0 = (c_0^1, c_0^2, \dots, c_0^{|\mathcal{T}|}, \underbrace{\perp, \perp, \dots, \perp}_{|\mathcal{L}|})$. Define global

action set $A = \bigcup_{i=1}^{|\mathcal{T}|} A_i$, global stack alphabet $\Gamma = \bigcup_{i=1}^{|\mathcal{T}|} \Gamma_i$. The global version of $\xrightarrow{a}$ is defined follows: $c \xrightarrow{a} c'$ with $c = (c^1, c^2, \dots, c^{|\mathcal{T}|}, \eta(l_1), \eta(l_2), \dots, \eta(l_{|\mathcal{L}|}))$, $c' = (c'^1, c'^2, \dots, c'^{|\mathcal{T}|}, \eta(l'_1), \eta(l'_2), \dots, \eta(l'_{|\mathcal{L}|}))$ where:

- $c \xrightarrow{a} c'$ where $a \in A$, if $\exists 1 \leq i \leq |\mathcal{T}|$, such that $c^i \xrightarrow{a} c'^i$ and $c^j = c'^j$ for all $j \neq i$ and $l_k = l'_k$ for $1 \leq k \leq |\mathcal{L}|$. This is the case that one of the threads performs an action.
- $c \xrightarrow{a(l_j)} c'$ if for all $1 \leq i \leq |\mathcal{T}|$, $c^i = c'^i$, $\exists 1 \leq j \leq |\mathcal{T}|$ such that $l_j = \perp$ and $l'_j \in [1, |\mathcal{L}|]$ and $l_k = l'_k$ for $k \neq j$. This is the case that thread $t_{l'_j}$ acquires lock l_j .
- $c \xrightarrow{\tau(l_j)} c'$ if for all $1 \leq i \leq |\mathcal{T}|$, $c^i = c'^i$, $\exists 1 \leq j \leq |\mathcal{T}|$ such that $l_j \in [1, |\mathcal{L}|]$ and $l'_j = \perp$ and $l_k = l'_k$ for $k \neq j$. This is the case that thread t_{l_j} releases lock l_j .

Lock accesses of P are said to be nested iff for any thread t in P , t only releases the most recently acquired lock among locks it holds.

Define $\mathfrak{L}(l) = \{\mathfrak{a}(l), \mathfrak{r}(l)\}$ to be the set of lock operations on lock l and $\mathfrak{L} = \bigcup_{l \in \mathcal{L}} \mathfrak{L}(l)$ to be the set of lock operations on all locks.

A sequence $\chi = x_1 x_2 \dots x_{k+1}$, where x 's are global configurations of P , is said to be an *execution* of P if $x_1 = c_0$ and $\exists a_1, a_2, \dots, a_k$ such that $\forall i$, $1 \leq i \leq k$, $x_i \xrightarrow{a_i} x_{i+1}$ where $a_i \in A \cup \mathfrak{L}$.

Definition 1. Given program P , lock set of thread t_i at global configuration c , denoted as $\mathfrak{S}(t_i, c)$, is the set of locks held by t_i at c , i.e. $\mathfrak{S}(t_i, c) = \{j | \eta(l_j) = i\}$ where $c = (c^1, c^2, \dots, c^{|\mathcal{T}|}, \eta(l_1), \eta(l_2), \dots, \eta(l_{|\mathcal{L}|}))$.

Definition 2. Let χ be the global execution of program P leading to global configuration c . For thread t_i , lock l_j such that $j \in \mathfrak{S}(t_i, c)$, define acquisition history $\mathcal{H}(t_i, l_j, c, \chi)$ to be set of indices of locks that were acquired (and possibly released) by t_i after the last acquisition of l_j by t_i along χ . Define acquisition history at configuration (state) c , $\mathcal{H}(t_i, c, \chi) = \bigcup_{l \in \mathfrak{S}(t_i, c)} \mathcal{H}(t_i, l, c, \chi)$.

Acquisition histories $\mathcal{H}(t_1, c_{o_1}, \chi_1)$ and $\mathcal{H}(t_2, c_{o_2}, \chi_2)$ are said to be compatible, denoted as $\mathcal{H}(t_1, c_{o_1}, \chi_1) \lesssim \mathcal{H}(t_2, c_{o_2}, \chi_2)$, if $\nexists l_1, l_2$ such that following conditions are true

- $l_1 \in \mathfrak{S}(t_1, c_{o_1})$

- $l_2 \in \mathfrak{S}(t_2, c_{o_2})$
- $l_1 \in \mathcal{H}(t_2, l_2, c_{o_2}, \chi_2)$
- $l_2 \in \mathcal{H}(t_1, l_2, c_{o_1}, \chi_1)$.

Using the above definitions, [7] presented an exact condition that characterizes whether a pair of threads can concurrently reach a pair of local states. The characterization is exploited by the algorithm used by PENELOPE, and is stated next. Here $\text{EF}(o_1 \wedge o_2)$ is a logical formula in temporal logic that describes the condition that local states o_1 and o_2 can be concurrently reached.

Theorem 1 (Pairwise reachability). *Given program P with threads t_1 and t_2 using nested locking, for states o_1 of t_1 and state o_2 of t_2 , $P \models \text{EF}(o_1 \wedge o_2)$ iff $\exists \chi_a, \chi_b$ such that χ_a leads t_1 to state o_1 , χ_b leads t_2 to state o_2 individually, and:*

- $\mathfrak{S}(t_1, c_{o_1}) \cap \mathfrak{S}(t_2, c_{o_2}) = \emptyset$
- $\mathcal{H}(t_1, c_{o_1}, \chi_1) \stackrel{c}{\sim} \mathcal{H}(t_2, c_{o_2}, \chi_2)$

CHAPTER 3

ALGORITHM OF PENELOPE

PENELOPE is a software framework that detects atomicity violations. It avoids the state space explosion problem by looking for typical violation patterns instead of trying all interleaves of instructions. PENELOPE checks for atomicity violations involving two threads and one variable, also known as *three-access atomicity violations*. PENELOPE uses an observed trace $\text{tr}(P)$ of program P as the start point, then tries to discover alternative schedulings of $\text{tr}(P)$ that lead to atomicity violations. The restriction of two threads and one variable is pragmatically determined and supported by the result of [1], which shows that a large number of atomicity violations in real world projects can be demonstrated with only two threads and one variable.

3.1 Formal Definition of Atomicity

Definition 3. *Given an execution $\chi = x_1x_2 \cdots x_{k+1}$, define execution trace tr corresponding to χ as the sequence of events $tr = e_1e_2 \cdots e_k$ where $x_1 \xrightarrow{e_1} x_2$, i.e. the i -th event e_i in tr makes program go from the i -th program state x_i to the $(i + 1)$ -th program state x_{i+1} .*

In addition to the definition given above, execution trace tr of a thread t_i is divided into *transactions*, by transaction start symbol $\triangleright_i$ and transaction end symbol $\triangleleft_i$. A transaction of t_i can be described by regular expression

$$\triangleright_i(A_i \cup \mathfrak{L})^* \triangleleft_i$$

$\triangleright_i$ and $\triangleleft_i$ do not change program states. Denote the k -th symbol in trace tr as $tr[k]$.

Let $|_t$ denote projection operation that yields data related to thread t only. For example denote the abstracted trace of observed run as tr , then $tr|_t$ is the

projection of abstracted trace including only actions of thread t from trace tr . Similarly, let $|_v$ denote projection operation resulting in data related to variable v only.

Definition 4. Two events e_1 and e_2 in program P are said to be dependent, denoted as $e_1 \stackrel{d}{\leftrightarrow} e_2$, if swapping the execution order of e_1 and e_2 leads P to be in a different program state; otherwise, if swapping does not change program state, then e_1 and e_2 are said to be independent, denoted as $e_1 \stackrel{d}{\nrightarrow} e_2$. Both $\stackrel{d}{\leftrightarrow}$ and $\stackrel{d}{\nrightarrow}$ are symmetric relations.

Definition 5. Two execution traces tr_1, tr_2 are said to be equivalent, denoted as $tr_1 \sim tr_2$ where $\sim$ is a symmetric relation, if either of the two following conditions is true:

- $tr_1 = tr_p e_1 e_2 tr_n, tr_2 = tr_p e_2 e_1 tr_n$ and $e_1 \stackrel{d}{\nrightarrow} e_2$
- $\exists tr_3$ such that $tr_1 \sim tr_3$ and $tr_2 \sim tr_3$

Definition 6. An execution trace is said to be serial if instruction interleaving happens at per transaction granularity rather than per instruction granularity. More specifically $\forall i, 1 \leq i \leq |\mathcal{T}|$, if $\triangleright_i = tr[p], \triangleleft_i = tr[q], p < q$ and $\nexists k$ s.t. $p < k < q, tr[k] = \triangleright_i \vee tr[k] = \triangleleft_i$, then $\forall j$ such that $p < j < q, tr[j] \in A_i \cup \mathfrak{L}$.

Definition 7. An execution trace tr is said to be atomic, iff $\exists tr'$ s.t. tr' is serial and $tr \sim tr'$.

3.2 Three-access Atomicity Violation

Three-access atomicity violation is a special case of atomicity violation involving two threads and three dependent events. Two of which, e_1 and e_2 , are from the same transaction of thread t_1 , the third, f , is from thread t_2 and occurs after e_1 but before e_2 . If $e_1 \stackrel{d}{\leftrightarrow} f$ and $e_2 \stackrel{d}{\leftrightarrow} f$ then events e_1, e_2, f are witnesses of a three-access atomicity violation.

For a given shared variable v , there are two classes of patterns for atomicity violation involving e_1, e_2 and f :

Definitions 4 to 6 borrowed from [8, 9]

- e_1 and e_2 being writes to v and f being a read to v .
- e_1 and e_2 being accesses (read or write) to v and f being a write to v .

The classes of patterns correspond to five violating patterns:

- $e_1:W, e_2:W, f:R$
- $e_1:R, e_2:R, f:W$
- $e_1:R, e_2:W, f:W$
- $e_1:W, e_2:R, f:W$
- $e_1:W, e_2:W, f:W$

[9] reduces the problem of three-access atomicity violation detection to *pairwise reachability* problem. The fact that f occurs after e_1 and before e_2 is equivalent to the fact that there is another event e , occurs in between e_1 and e_2 in the same transaction, that can be scheduled to run concurrently with f , i.e. it should be possible for program execution to reach e and f in parallel.

Definition 8 ([9]). *Given two threads t_i, t_j and a shared variable $v \in \mathcal{V}$, there is a three-access atomicity violation iff there are three events $e_1 = tr_i[k_1], e_2 = tr_i[k_2], f = tr_j[k_3]$ s.t.:*

- e_1 and e_2 are in the same transaction
- $e_1 \xleftrightarrow{d} f$ and $e_2 \xleftrightarrow{d} f$
- $\exists e = tr_i[k_4], k_1 \leq k_4 < k_2$ s.t. e and f are pairwise reachable from the start of program

With theorem 1 and definition 8, to know if two events e, f are witnesses of three-access atomicity violation, we need to know:

- whether e and f are both individually reachable in the corresponding thread
- what the value of lock sets and acquisition histories are

PENELOPE uses an actual trace as the start point of analysis, all events observed are proven to be reachable, so the problem left is how to get lock sets and acquisition histories at these observed states. PENELOPE proposed a solution that runs in time linear to the length of trace file, in fact a one-pass algorithm. An implementation is shown below (incorporating idea of *last acquisition* that will be covered in chapter 5), where $\mathfrak{S}$ represents for lock set and $\mathcal{A}$ represents for last acquisition:

```

PENELOPELOCKSETANDLASTACQUISITION( $\text{tr} \mid_t, \mathcal{I}$ ):
 $\mathcal{A} \leftarrow \emptyset$ 
 $\mathfrak{S} \leftarrow \emptyset$ 
for  $w_i$  in  $\text{tr} \mid_t = w_1 w_2 \cdots w_{|\text{tr} \mid_t|}$ :
    if  $w_i = \mathfrak{a}(l)$  for some  $l \in \mathcal{L}$ :
         $\mathfrak{S} \leftarrow \mathfrak{S} \cup \{l\}$ 
         $\mathcal{A}(l) \leftarrow i$ 
    if  $w_i = \mathfrak{r}(l)$  for some  $l \in \mathcal{L}$ :
         $\mathfrak{S} \leftarrow \mathfrak{S} \setminus \{l\}$ 
    if  $i \in \mathcal{I}$ :
         $\mathfrak{S}(i) \leftarrow \mathfrak{S}$ 
         $\mathcal{A}(i) \leftarrow \mathcal{A}$ 
return  $\mathfrak{S}, \mathcal{A}$ 

```

With careful implementation, the work of getting R, W, WW, AA sets and computing lock sets and acquisition histories of set elements can be done with one pass. PENELOPELOCKSETANDACQUISITION runs in $O(|\text{tr}| \cdot |\mathcal{L}|)$ time with $O(|\text{tr}| \cdot |\mathcal{L}|)$ space.

After knowing value of lock sets and acquisition histories of all events of interest, atomicity violation detection is check by algorithm below, following theorem 1.

```

FINDVIOLATIONS( $\mathcal{V}, \mathcal{T}$ ):
 $P \leftarrow \{(\text{WW}, \text{R}), (\text{RR}, \text{W}), (\text{RW}, \text{W}), (\text{WR}, \text{W}), (\text{WW}, \text{W})\}$ 
for  $v$  in  $\mathcal{V}$ :
    for  $(E, F)$  in  $P$ :
        for  $(t_i, t_j)$  s.t.  $t_i, t_j \in \mathcal{T}$  and  $i \neq j$ :
            for  $e \in E \mid_{t_i, v}, f \in F \mid_{t_j, v}$ :
                if  $\mathfrak{S}(t_i, e) \cap \mathfrak{S}(t_j, f) = \emptyset \wedge \mathcal{H}(t_i, e) \stackrel{\mathcal{L}}{\sim} \mathcal{H}(t_j, f)$ :
                    Atomicity violation found

```

In code above, e, f are configurations when read/write actions happen. χ 's are omitted from $\mathcal{H}$ since PENELOPE only observes one execution for any thread thus execution is not a variable.

CHAPTER 4

STRAIGHT LINE GRAMMAR

Definition 9. A straight line grammar (SLG) is a special class of context free grammar (CFG). Formally, SLG is a tuple $(\Sigma, \Pi, \mathcal{P}, S)$, where Σ is the set of terminal symbols, Π is the set of non-terminal symbols, $\Sigma \cap \Pi = \emptyset$, $\mathcal{P} : \Pi \rightarrow (\Sigma \cup \Pi)^*$ is the production rule function and $S \in \Pi$ is the start symbol. $\forall \pi \in \Pi, \pi \notin \mathcal{P}(\pi)$. Define relation $\hookrightarrow \in \Pi \times \Pi$, $\pi_1 \hookrightarrow \pi_2$ if either of the following is true:

- $\pi_2 \in \mathcal{P}(\pi_1)$
- $\exists \pi_3$ such that $\pi_1 \hookrightarrow \pi_3$ and $\pi_3 \hookrightarrow \pi_2$

There exists a permutation $\delta(\Pi) = \{\delta_1, \delta_2, \dots, \delta_{|\Pi|}\}$ such that $\nexists 1 \leq i < j \leq |\Pi|, \delta_j \hookrightarrow \delta_i$.

Remark 1. $\mathcal{P}$ is a function, rather than a relation as in definition of general context free grammar. That is, for SLG a nonterminal symbol has exactly one derivation rule, while for CFG a nonterminal symbol can have more than one derivation rules.

For SLG a nonterminal symbol has exactly one string it can derive to, which should not contain itself. More over there exists an acyclic dependency between nonterminals. Without loss of generality, in this thesis it is assumed that there are *exactly two* symbols on the right hand side of any production rule. In the rest of this thesis, for nonterminal symbol s , $\mathcal{P}(s)$ is considered as a list, and list operations borrowed from functional programming are applied.

Definition 10. A word $w \in (\Sigma \cup \Pi)^*$ is a sequence of terminal and non-terminal symbols. The length of a word w is the number of symbols in w , denoted as $|w|$. Each symbol in w is indexed by its position, starting from 1, $w = w_1 w_2 \dots w_{\ell(w)}$. A symbol s is contained in w , denoted as $s \in w$, iff $\exists i$ such that $1 \leq i \leq \ell(w), s = w_i$. Given a nonterminal symbol π , the string

containing only terminal symbols produced following the production rules from π is denoted as $w_{\pi \rightsquigarrow}$, $\forall \pi \in \Pi$, $w_{\pi \rightsquigarrow} \in \Sigma^*$. For terminal symbol $\sigma \in \Sigma$, define $w_{\sigma \rightsquigarrow} = \sigma$.

For word w , the substring starting from index i to index j is denoted as $w[i, j]$. In particular, $w = w[1, |w|]$.

Definition 11. Given SLG $g = (\Sigma, \Pi, \mathcal{P}, S)$, height of symbol s , denoted as $h(s)$, is defined as follows (does not apply to start symbol S):

$$h(s) = \begin{cases} 0 & s \in \Sigma \\ 1 + \max(h(s_1), h(s_2)) & \mathcal{P}(s) = s_1 s_2 \end{cases}$$

Definition 12. Given SLG $g = (\Sigma, \Pi, \mathcal{P}, S)$, length of symbol $s \in \Sigma \cup \Pi$, denoted as $\ell(s)$, is the length of string w that s fully decompresses to, i.e. $\ell(w_{s \rightsquigarrow})$. The recursive definition of ℓ is:

$$\ell(s) = \begin{cases} 1 & s \in \Sigma \\ \ell(s_1) + \ell(s_2) & \mathcal{P}(s) = s_1 s_2 \end{cases}$$

The string encoded by SLG string can be reconstructed by doing invocation of $\text{DERIVE}(S, \mathcal{P})$.

```

DERIVE( $s, \mathcal{P}$ ):
  if  $s \in \Sigma$ :
    return  $[s]$ 
   $w_{s \rightsquigarrow} \leftarrow \text{NIL}$ 
  for  $i$  in 1 to  $|\mathcal{P}(s)|$ , where  $\mathcal{P}(s) = s_1 s_2 \cdots s_{|\mathcal{P}(s)|}$ :
     $w_{s \rightsquigarrow} \leftarrow \text{concat}(w_{s \rightsquigarrow}, \text{DERIVE}(s_i, \mathcal{P}))$ 
  return  $w_{s \rightsquigarrow}$ 

```

CHAPTER 5

ALGORITHM ON GRAMMAR COMPRESSED TRACE

In addition to saving storage space, compression can be considered as a way of exploiting repeated patterns in input, thus finding computation results that can be cached and reused. In this chapter it is assumed in addition to trace tr , there is a grammar compressed trace available, denoted as ctr .

5.1 Lock Value Difference and Last Acquisition

Definition 13. Given SLG $g = (\Sigma, \Pi, \mathcal{P}, S)$, last acquisition of lock l on string $w = w_1w_2 \cdots w_{|w|}$, where $w_i \in \Sigma$ is a function $\bar{\Lambda} : \mathcal{L} \times \Sigma^* \rightarrow \mathbb{N}$, returning 1-based index of last lock acquisition operation $\mathbf{a}(l)$ in w , or 0 if $\mathbf{a}(l) \notin w$.

$$\bar{\Lambda}(l, w) = \begin{cases} 0 & \text{if } \nexists i \text{ such that } w_i = \mathbf{a}(l) \\ \max\{i \mid 1 \leq i \leq |w| \wedge w_i = \mathbf{a}(l)\} & \text{otherwise} \end{cases}$$

We can define last acquisition for string w compressed by symbol $s \in \Sigma \cup \Pi$, i.e. $w = w_{s\rightsquigarrow}$, as the last acquisition for symbol s :

Definition 14. Last acquisition of lock l on symbol s for SLG $g = (\Sigma, \Pi, \mathcal{P}, S)$ is a function $\Lambda : \mathcal{L} \times (\Sigma \cup \Pi) \rightarrow \mathbb{N}$, returning 1-based index of last acquisition of l in string $w_{s\rightsquigarrow}$, or 0 if $\mathbf{a}(l) \notin w_{s\rightsquigarrow}$.

$$\Lambda(l, s) = \begin{cases} 1 & \text{if } s = \mathbf{a}(l) \\ 0 & \text{if } s \neq \mathbf{a}(l) \wedge s \in \Sigma \\ \ell(s_1) + \Lambda(l, s_2) & \text{if } \mathcal{P}(s) = s_1s_2 \wedge \Lambda(l, s_2) \neq 0 \\ \Lambda(l, s_1) & \text{if } \mathcal{P}(s) = s_1s_2 \wedge \Lambda(l, s_2) = 0 \end{cases}$$

$\hat{\Lambda} : \mathcal{L} \times (\Sigma \cup \Pi) \times \mathbb{N} \rightarrow \mathbb{N}$ can be defined by generalizing Λ and take an index as extra parameter. $\hat{\Lambda}(l, s, \text{idx})$ will return the last acquisition value of substring $w_{s \rightsquigarrow}[1, \text{idx}]$. In particular, $\Lambda(l, s) = \hat{\Lambda}(l, s, \ell(s))$.

```

 $\hat{\Lambda}(l, s, \text{idx}):$ 
  if  $\text{idx} = \ell(s)$ :
    return  $\Lambda(l, s)$ 
  if  $\hat{\Lambda}(l, s_2, \text{idx} - \ell(s_1)) > 0$ :
    return  $\ell(s_1) + \hat{\Lambda}(l, s_2, \text{idx} - \ell(s_1))$ 
  else:
    return  $\Lambda(l, s_1)$ 

```

Definition 15. For program with a single thread t , let $\zeta(l)$ be the status value of lock l , where $\zeta(l) = 1$ when locked by t and $\zeta(l) = 0$ when unlocked. Lock value difference of a string $w = w_1 w_2 \cdots w_{|w|}$, where $w_i \in \Sigma$, is a function $\bar{\Delta} : \mathcal{L} \times \Sigma^* \rightarrow \{-1, 0, 1\}$. $\bar{\Delta}(l, \mathbf{a}(l))$ is defined to be 1, $\bar{\Delta}(l, \mathbf{r}(l))$ is defined to be -1 and $\bar{\Delta}(l, a)$, where $a \in A_i$, is defined to be 0. $\bar{\Delta}(l, w) = \sum_{i=1}^{|w|} \bar{\Delta}(l, w_i)$

Remark 2. $\bar{\Delta} : \mathcal{L} \times \Sigma^* \rightarrow \{-1, 0, 1\}$ since there are only two possible state values, and $\text{tr}|_t$ is assumed to respect lock behavior that an acquired lock can not be acquired and a free lock can not be released.

For example given thread t and its abstracted trace $\text{tr}|_t$ with first i -th events being consumed. The current state of t corresponds to the configuration after the first i events happen. Let us say the current status value for lock l is 1, meaning t currently holds l . If $\bar{\Delta}(l, \text{tr}|_t[i+1, j]) = 0$, where $j \geq i+1$, i.e. the status value of l is not changed by events $i+1$ through j , meaning the status value of l by the end of event j is the same as the status value of l by the end of event i . As a result, t still holds l by the end of event j , even though l might be released and reacquired by t .

We can define lock value difference for string w compressed by symbol $s \in \Sigma \cup \Pi$, i.e. $w = w_{s \rightsquigarrow}$, as the lock value difference for symbol s :

Definition 16. Lock value difference of symbol $s \in \Sigma \cup \Pi$ for lock l and SLG $g = (\Sigma, \Pi, \mathcal{P}, S)$, denoted as $\Delta(l, s)$, is the difference of l 's state value after consuming symbol s :

$$\Delta(l, s) = \begin{cases} 1 & s = \mathbf{a}(l) \\ -1 & s = \mathbf{r}(l) \\ 0 & s \notin \mathfrak{L}(l) \wedge s \in \Sigma \\ \Delta(l, s_1) + \Delta(l, s_2) & \mathcal{P}(s) = s_1 s_2 \end{cases}$$

Acquisition histories can be computed, given lock sets and last acquisitions (shown in section 5.3). Similar to how Λ is generalized to $\hat{\Lambda}$, we can generalize Δ to $\hat{\Delta}$ to compute lock value difference of any prefix of the string compressed by symbol s :

```

 $\hat{\Delta}(l, s, \text{idx}):$ 
  if  $\text{idx} = \ell(s)$ :
    return  $\Delta(l, s)$ 
  if  $\text{idx} \leq \ell(s_1)$ :
    return  $\hat{\Delta}(l, s_1, \text{idx})$ 
  else:
    return  $\Delta(l, s_1) + \hat{\Delta}(l, s_2, \text{idx} - \ell(s_1))$ 

```

For the rest of this thesis, for a given lock l , we use Λ_l to denote both Λ and $\hat{\Lambda}$, use Δ_l to denote both Δ and $\hat{\Delta}$. The function actually used can be determined by number of parameters being supplied.

Definitions above are per thread properties, when multiple threads are involved, thread ID can be used, for example, to distinguish last acquisition function of thread t_1 from that of thread t_2 . Compression of abstract trace is done at per thread level, as intuition suggests that trace per thread is more compressible than that of the whole program, which can be an arbitrary interleaving of individual thread traces. This, however, is a conjecture not proven in experiment and is listed as future work.

With definitions 14 and 16 and their corresponding generalization, we can compute $\mathfrak{S}$ and $\mathcal{A}$ at the specific index of trace.

```

LOCKSETAT(ctr |t, idx):
  for  $l$  in  $\mathcal{L}$ :
     $\zeta(l) \leftarrow 0$ 
  for  $i$  in 1 to |ctr |t|, where  $\text{ctr } |_t = w_1 w_2 \cdots w_{|\text{ctr } |_t|}$ :
    if  $\ell(w_i) < \text{idx}$ :
       $\text{idx} \leftarrow \text{idx} - \ell(w_i)$ 
      for  $l$  in  $\mathcal{L}$ :
         $\zeta(l) \leftarrow \zeta(l) + \Delta_l(w_i)$ 
    else:
      for  $l$  in  $\mathcal{L}$ :
         $\zeta(l) \leftarrow \zeta(l) + \Delta_l(w_i, \text{idx})$ 
      break
  return  $\{l | \zeta(l) = 1\}$ 

```

```

LASTACQUISITIONAT(ctr |t, idx):
  for  $l$  in  $\mathcal{L}$ :
     $\mathcal{A}(l) \leftarrow 0$ 
   $c \leftarrow 0$ 
  for  $i$  in 1 to |ctr |t|, where  $\text{ctr } |_t = w_1 w_2 \cdots w_{|\text{ctr } |_t|}$ :
    if  $\ell(w_i) < \text{idx}$ :
      for  $l$  in  $\mathcal{L}$ :
        if  $\Lambda_l(w_i) > 0$ :
           $\mathcal{A}(l) \leftarrow c + \Lambda_l(w_i)$ 
         $c \leftarrow c + \ell(w_i)$ 
       $\text{idx} \leftarrow \text{idx} - \ell(w_i)$ 
    else:
      for  $l$  in  $\mathcal{L}$ :
        if  $\Lambda_l(w_i, \text{idx}) > 0$ :
           $\mathcal{A}(l) \leftarrow c + \Lambda_l(w_i, \text{idx})$ 
        break
  return  $\mathcal{A}$ 

```

5.2 Computing Indices for Events of Interest

We need to track four sets of events of interest: $\mathcal{R}$, $\mathcal{W}$, $\mathcal{WW}$, $\mathcal{AA}$ where no set should contain two events with the same $(\mathfrak{S}, \mathcal{A})$ pair. First we have the following observation:

Observation 1. $(\mathfrak{S}, \mathcal{A})$ pair changes iff at least one lock operation is encountered

Now we know that we only need to track at most one event of a kind upon seeing a lock operation. Functions computing indices for events of interest in $O(|\text{tr}|)$ time with $O(1)$ additional space are shown below.

Function READINDICES takes projection of uncompressed trace to a thread t , and outputs the index set of read events.

```

READINDICES(tr):
   $\mathcal{I} \leftarrow \text{NIL}$ 
   $b \leftarrow \top$ 
  for  $i$  in 1 to  $|\text{tr}|$ , where  $\text{tr} = w_1 w_2 \cdots w_{|\text{tr}|}$ :
    if  $w_i = \text{R}(v)$  for some  $v \in \mathcal{V}$  and  $b = \top$ :
       $\mathcal{I} \leftarrow \mathcal{I} \text{snoc}(v, i)$ 
       $b \leftarrow \perp$ 
    if  $w_i = \triangleright$  or  $w_i \in \mathfrak{L}$ :
       $b \leftarrow \top$ 
  return  $\mathcal{I}$ 

```

Function WRITEINDICES computing indices of write events can be obtained by modifying the if condition in READINDICES from

$$w_i = \text{R}(v)$$

to

$$w_i = \text{W}(v)$$

Function WRITEWRITEINDICES computes indices of events sandwiched between two write events.

```

WRITEWRITEINDICES(tr):
   $\mathcal{I} \leftarrow \text{NIL}$ 
  PREV  $\leftarrow \text{NIL}$ 
   $b \leftarrow \top$ 
  for  $i$  in 1 to  $|\text{tr}|$ , where  $\text{tr} = w_1 w_2 \cdots w_{|\text{tr}|}$ :
    if  $w_i = \text{W}(v)$  for some  $v \in \mathcal{V}$ :
      if PREV  $\neq \text{NIL}$ :
         $\mathcal{I} \leftarrow \mathcal{I} \text{ snoc } \text{PREV}$ 
        PREV  $\leftarrow \text{NIL}$ 
      if  $b = \top$ :
        PREV  $\leftarrow (v, i)$ 
         $b \leftarrow \perp$ 
    if  $w_i = \triangleright$  or  $w_i \in \mathfrak{L}$ :
       $b \leftarrow \top$ 
    if  $w_i = \triangleleft$ :
      PREV  $\leftarrow \text{NIL}$ 
  return  $\mathcal{I}$ 

```

Similar to how WRITEINDICES can be obtained through modifying READINDICES, function ACCESSACCESSINDICES computing indices of events sandwiched between two variable accesses (read or write) can be obtained by modifying the if condition in WRITEWRITEINDICES from

$$w_i = \text{W}(v)$$

to

$$w_i = \text{R}(v) \text{ or } w_i = \text{W}(v)$$

Notice that returned index set $\mathcal{I}$ is sorted for all four functions. $\mathcal{I}|_t$ can be obtained in linear time by merging four sorted index sets with duplicate removal.

Remark 3. In $\mathcal{I}|_t$ constructed above, no two events of the same kind e, e' have identical $(\mathfrak{S}, \mathcal{A})$ pair but it is possible for e, e' to have identical $(\mathfrak{S}, \mathcal{H})$ pair. This is not a problem as redundant input will be discarded by the algorithm run in next stage.

5.3 Getting Lock Sets and Acquisition Histories for All Events

Algorithm below is an adoption of LOCKSETAT and LASTACQUISITIONAT, taking a list of indices as input as opposed to one single index. Two mappings are returned: $\mathcal{M}_{\mathfrak{S}}$ maps event indices to the corresponding lock sets and $\mathcal{M}_{\mathcal{A}}$ maps event indices to corresponding last acquisitions.

```

LOCKSETSANDLASTACQUISITION( $\text{ctr}|_t, \mathcal{I}|_t$ ):
  for  $l$  in  $\mathcal{L}$ :
     $\zeta(l) \leftarrow 0$ 
     $\mathcal{A}(l) \leftarrow 0$ 
   $c \leftarrow 0$ 
   $\mathcal{M}_{\mathfrak{S}} \leftarrow \emptyset$ 
   $\mathcal{M}_{\mathcal{A}} \leftarrow \emptyset$ 
  for  $i$  in 1 to  $|\text{ctr}|_t|$ , where  $\text{ctr}|_t = w_1 w_2 \cdots w_{|\text{ctr}|_t}$ :
    while  $\text{car}(\mathcal{I}|_t) < c + \ell(w_i)$ :
       $\text{idx} \leftarrow \text{car}(\mathcal{I}|_t) - c$ 
       $\mathcal{A}' \leftarrow \mathcal{A}$ 
      for  $l$  in  $\mathcal{L}$ :
         $\zeta'(l) \leftarrow \zeta(l) + \Delta_l(w_i, \text{idx})$ 
        if  $\Lambda_l(w_i, \text{idx}) > 0$ :
           $\mathcal{A}'(l) \leftarrow c + \Lambda_l(w_i, \text{idx})$ 
       $\mathcal{M}_{\mathfrak{S}}(\text{car}(\mathcal{I}|_t)) \leftarrow \{l \mid \zeta'(l) = 1\}$ 
       $\mathcal{M}_{\mathcal{A}}(\text{car}(\mathcal{I}|_t)) \leftarrow \text{SORT}(\mathcal{A}')$ 
       $\mathcal{I}|_t \leftarrow \text{cdr}(\mathcal{I}|_t)$ 
    for  $l$  in  $\mathcal{L}$ :
       $\zeta(l) \leftarrow \zeta(l) + \Delta_l(w_i)$ 
      if  $\Lambda_l(w_i) > 0$ :
         $\mathcal{A}(l) \leftarrow c + \Lambda_l(w_i)$ 
     $c \leftarrow c + \ell(w_i)$ 
  return  $\mathcal{M}_{\mathfrak{S}}, \mathcal{M}_{\mathcal{A}}$ 

```

Remark 4. Notice $\mathcal{A}$ is sorted in ascending order of last acquisition value before stored, as we only care about the relative ordering of past lock acquisition.

LOCKSETSANDLASTACQUISITION runs in $O(|\text{ctr}|_t + |\mathcal{L}| \cdot |\mathcal{I}|_t \cdot h_{\max})$ time with $O(|\mathcal{L}| \cdot |\Pi|)$ space, where $h_{\max} = \max_{\pi \in \Pi} h(\pi)$. The running time and space

requirement only depend on the length of *compressed* trace and properties of the grammar compressing the trace.

Given two sorted last acquisitions $\mathcal{A}_1, \mathcal{A}_2$ and lock sets $\mathfrak{S}_1, \mathfrak{S}_2$, we can use algorithm below to determine if corresponding acquisition histories are compatible in $O(|\mathcal{L}|)$ time with $O(1)$ additional space.

```

ISCOMPATIBLE( $\mathfrak{S}_1, \mathcal{A}_1, \mathfrak{S}_2, \mathcal{A}_2$ ):
  while car( $\mathcal{A}_1$ ) not in  $\mathfrak{S}_1$ :
     $\mathcal{A}_1 \leftarrow \text{cdr}(\mathcal{A}_1)$ 
  while car( $\mathcal{A}_2$ ) not in  $\mathfrak{S}_2$ :
     $\mathcal{A}_2 \leftarrow \text{cdr}(\mathcal{A}_2)$ 
   $\mathcal{A}_1 \leftarrow \text{filter}(l \in \mathfrak{S}_1 \vee l \in \mathfrak{S}_2, \mathcal{A}_1)$ 
   $\mathcal{A}_2 \leftarrow \text{filter}(l \in \mathfrak{S}_1 \vee l \in \mathfrak{S}_2, \mathcal{A}_2)$ 
  while  $\mathcal{A}_1 \neq \text{NIL}$  and  $\mathcal{A}_2 \neq \text{NIL}$ :
    if car( $\mathcal{A}_1$ )  $\neq$  car( $\mathcal{A}_2$ ):
      return false
     $\mathcal{A}_1 \leftarrow \text{cdr}(\mathcal{A}_1)$ 
     $\mathcal{A}_2 \leftarrow \text{cdr}(\mathcal{A}_2)$ 
  return true

```

CHAPTER 6

SELECTION OF STRAIGHT LINE GRAMMAR

The algorithm described in chapter 5 works for any grammar based codes whose production rules always yield exactly two symbols. When it comes to implementation, we need to have a grammar based compression algorithm that both produces small sized grammar and runs in reasonable amount of time. We decided to implement **Sequitur** algorithm for our experiment.

Sequitur is an algorithm that does grammar based compression in linear time in the size of uncompressed file [4]. An advantage of **Sequitur** is that it works incrementally and can take inputs in a stream. It is shown in [10] that with appropriate encoding, **Sequitur** has a compression rate comparable to that of popular compression tools such as **gzip**, **compress** and **PPMC**.

Sequitur algorithm yields a SLG with the following additional two properties:

- Digram uniqueness: No pair of adjacent symbols (digram) appears more than once in the grammar.
- Rule utility: Every production rule is used more than once.

Notice there can possibly be multiple SLG's satisfying these two properties for a given input, **Sequitur** yields one of them. The basic idea of **Sequitur** is tracking all digrams currently in grammar, replace a digram with a symbol whenever the digram reoccurs in grammar, creating a production rule from the new symbol to the replaced digram. As a result, each nonterminal symbol, excluding special start symbol S , should occur in grammar at least twice. If a nonterminal symbol appears only once, the corresponding production rule will be removed and nonterminal symbol will be replaced by its target string. This ensures no unnecessary production rules in grammar.

Below is the pseudo code of **Sequitur** that produces grammar whose production rules have exactly 2 symbols on the right hand side. In the code

$\mathcal{D} : \Sigma \cup \Pi \times \Sigma \cup \Pi \rightarrow \Pi \times \mathbb{N}$ is the digram map. $\mathcal{D}$ maps a pair of symbols to a nonterminal symbol and a occurrence count.

```

DIGRAMUNIQUENESS( $\pi, (s_1, s_2), \varpi$ ):
  concat( $l_p, s_1$  cons  $s_2$  cons  $l_n$ )  $\leftarrow \mathcal{P}(\pi)$ 
  if  $l_p \neq \text{NIL}$ :
     $l'_p$  snoc  $s_p \leftarrow l_p$ 
     $(\pi', c) \leftarrow \mathcal{D}(s_p, s_1)$ 
     $\mathcal{D}(s_p, s_1) \leftarrow (\pi', c - 1)$ 
  if  $l_n \neq \text{NIL}$ :
     $s_n$  cons  $l'_n \leftarrow l_n$ 
     $(\pi', c) \leftarrow \mathcal{D}(s_2, s_n)$ 
     $\mathcal{D}(s_2, s_n) \leftarrow (\pi', c - 1)$ 
   $\mathcal{P}(\pi) \leftarrow \text{concat}(l_p, \varpi \text{ cons } l_n)$ 

```

```

SEQUITURCOMPRESS( $w$ ):
   $\mathcal{P}(S) \leftarrow [w_1]$ 
  for  $i$  in 2 to  $|w|$ , where  $w = w_1 w_2 \cdots w_{|w|}$ :
     $\mathcal{P}(S)$  snoc  $s \leftarrow \mathcal{P}(S)$ 
     $n \leftarrow w_i$ 
    while  $(s, n)$  in  $\mathcal{D}$ :
       $(\pi, c) \leftarrow \mathcal{D}(s, w_i)$ 
      if  $c = 1$ :
         $\varpi \leftarrow \text{NEWSYMBOL}()$ 
        DIGRAMUNIQUENESS( $\pi, (s, n), \varpi$ )
         $\mathcal{D}(s, n) \leftarrow (\varpi, 2)$ 
         $\mathcal{P}(S) \leftarrow \mathcal{P}(S)$  snoc  $\varpi$ 
      else:
         $\mathcal{D}(s, n) \leftarrow (\pi, c + 1)$ 
         $\mathcal{P}(S) \leftarrow \mathcal{P}(S)$  snoc  $\pi$ 
      if  $|\mathcal{P}(S)| > 1$ :
         $\mathcal{P}(S)$  snoc  $s$  snoc  $n \leftarrow \mathcal{P}(S)$ 
      else:
        continue to next for loop iteration
     $\mathcal{D}(s, n) \leftarrow (S, 1)$ 
     $\mathcal{P}(S) \leftarrow \mathcal{P}(S)$  snoc  $s$  snoc  $n$ 
  return  $\mathcal{P}$ 

```

For **Sequitur** algorithm, although worst case running time for single insertion is $O(\sqrt{n})$, the amortized running time is $6n = O(n)$. The overall

compression runs in time linear in the input size. Recall that `READINDICES` and `WRITEWRITEINDICES` both run in time linear in the input size. Asymptotically, `Sequitur` does not add any extra overhead to preprocessing.

CHAPTER 7

EXPERIMENTS

We used the same applications and test harnesses as the ones used by **PENELOPE**. All applications are written in **Java** where lock operations are nested through **synchronized** keyword. Comparisons between algorithms with and without grammar based compression are made.

For **elevator**, and **tsp**, the input files were included in the benchmarks, and the table indicates which input file was used for the results. For **elevator**, the number of threads was also specified in the input files, and there were no additional parameters to be provided by the user. The test harness for **tsp** includes an input file, a given number of threads, and a script would compare the minimum tour computed by the program against the minimum tour computed by a single thread execution.

For **Vector (Stack)**, test harnesses with two threads and two small vectors (stacks) are provided, where each thread executes exactly one method from class **Vector (Stack)**.

Sequitur and algorithms in chapter 5 are implemented in **Python**, running on **CPython** interpreter. Experiments are run on machine with quadcore 2.8GHz Intel i7 CPU, 16GB of RAM and 512GB of SSD.

In table 7.1, trace length and compressed trace length are measured by number of symbols. Nonterminals is the number of nonterminal symbols in **Sequitur** grammar, we use it here as a measure of grammar size. Running time for compressed algorithm includes time of both trace compression and atomicity detection.

Table 7.1: Experiment Results

Application (LOC)	Input	Threads	Variables	Locks	Trace Length	Compressed Trace Length	Nonterminals	Compression Rate	Time (Uncompressed)	Time (Compressed)
Vector (1.3K)	VectorTest	4	12	2	142	83	29	59%	0.01s	0.01s
	VectorTest1	4	12	2	231	122	53	53%	0.01s	0.01s
	VectorTest2	4	12	2	248	131	44	53%	0.01s	0.01s
	VectorTest3	4	12	2	231	95	46	41%	0.01s	0.01s
	VectorTest4	4	12	2	159	86	33	54%	0.01s	0.01s
Stack (1.4K)	StackTest	4	12	2	136	90	10	66%	0.01s	0.01s
	StackTest1	4	12	2	209	125	38	60%	0.01s	0.01s
	StackTest2	4	12	2	231	132	50	57%	0.01s	0.01s
	StackTest3	4	12	2	265	140	41	53%	0.01s	0.01s
	StackTest4	4	12	2	127	77	13	61%	0.01s	0.01s
elevator (566)	data	3	65	8	26915	8670	1179	31%	10.71s	8.71s
	data2	5	113	8	54566	13359	2511	24%	7.12s	3.64s
	data3	5	457	50	329913	69757	18089	21%	12m49s	3m18s
tsp (794)	map13	2	588	2	32250186	6116075	712365	19%	4m05s	1m40s
	map14	4	652	2	14026944	3132490	489379	22%	3m21s	42s

CHAPTER 8

CONCLUSION

In this paper, we presented an improved algorithm detecting atomicity violation based on **PENELOPE**. We captured two functions, lock value difference and last acquisition, that can be calculated directly on grammar compressed trace. In our algorithm, compression is more than a method to reduce storage space, but also as a preprocessing stage for later computation. The algorithm presented in chapter 5 works for any grammar based compression algorithm. We chose to implement **Sequitur** due to its simplicity and incremental construction. Our algorithm overall runs in time linear in the length of compressed program execution trace and length of longest derivation path in grammar, with space linear in the size of grammar. The improvement is significant especially with long execution traces as the compression rate in these cases tend to be high.

REFERENCES

- [1] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” *SIGPLAN Not.*, vol. 43, no. 3, pp. 329–339, Mar. 2008. [Online]. Available: <http://doi.acm.org/10.1145/1353536.1346323>
- [2] F. Sorrentino, A. Farzan, and P. Madhusudan, “Penelope: Weaving threads to expose atomicity violations,” in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE ’10. New York, NY, USA: ACM, 2010. [Online]. Available: <http://doi.acm.org/10.1145/1882291.1882300> pp. 37–46.
- [3] J. C. Kieffer and E.-H. Yang, “Grammar-based codes: A new class of universal lossless source codes,” *IEEE Trans. Inf. Theor.*, vol. 46, no. 3, pp. 737–754, Sep. 2006. [Online]. Available: <http://dx.doi.org/10.1109/18.841160>
- [4] C. G. Nevill-manning and I. H. Witten, “Inferring sequential structure,” Tech. Rep., 1996.
- [5] N. J. Larsson and A. Moffat, “Offline dictionary-based compression,” in *Proceedings of the Conference on Data Compression*, ser. DCC ’99. Washington, DC, USA: IEEE Computer Society, 1999. [Online]. Available: <http://dl.acm.org/citation.cfm?id=789086.789701> pp. 296–.
- [6] C. G. Nevill-Manning and I. H. Witten, “Identifying hierarchical structure in sequences: A linear-time algorithm,” *J. Artif. Int. Res.*, vol. 7, no. 1, pp. 67–82, Sep. 1997. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1622776.1622780>
- [7] V. Kahlon, F. Ivančić, and A. Gupta, “Reasoning about threads communicating via locks,” in *Proceedings of the 17th International Conference on Computer Aided Verification*, ser. CAV’05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: http://dx.doi.org/10.1007/11513988_49 pp. 505–518.

- [8] C. Wang, R. Limaye, M. Ganai, and A. Gupta, “Trace-based symbolic analysis for atomicity violations,” in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS’10. Berlin, Heidelberg: Springer-Verlag, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_27 pp. 328–342.
- [9] A. Farzan, P. Madhusudan, and F. Sorrentino, “Meta-analysis for atomicity violations under nested locking,” in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV ’09. Berlin, Heidelberg: Springer-Verlag, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_21 pp. 248–262.
- [10] C. G. Nevill-manning and I. H. Witten, “Compression and explanation using hierarchical grammars,” *Computer Journal*, vol. Vol. 40, p. 103, 1997.