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 PENEOPE ................................. 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: $P_i = (L_i, A_i, \Gamma_i, c_i^0, T_i)$. $L_i$ 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); $\Gamma_i$ 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. $c_i^0$ 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 $v \in \Gamma^*$.

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

A multi-threaded program $P$ with $|T|$ threads and $|\mathcal{L}|$ locks can be represented as a tuple $P = (P_1, P_2, \ldots, P_{|T|}, \eta(l_1), \eta(l_2), \ldots, \eta(l_{|\mathcal{L}|}))$ whose initial state, or global initial state $c_0 = (c_1^0, c_2^0, \ldots c_{|T|}^0, \perp, \perp, \ldots, \perp)$. Define global
action set $A = \bigcup_{i=1}^{|T|} A_i$, global stack alphabet $\Gamma = \bigcup_{i=1}^{|T|} \Gamma_i$. The global version of $\xrightarrow{a}$ is defined follows: $c \xrightarrow{a} c'$ with $c = (c^1, c^2, \ldots, c^{|T|}, \eta(l_1), \eta(l_2), \ldots, \eta(l_{|L|}))$, $c' = (c'^1, c'^2, \ldots, c'^{|T|}, \eta(l'_1), \eta(l'_2), \ldots, \eta(l'_{|L|}))$ where:

- $c \xrightarrow{a} c'$ where $a \in A$, if $\exists 1 \leq i \leq |T|$, such that $c^i \xrightarrow{a} c'^i$ and $c^j = c'^j$ for all $j \neq i$ and $l_k = l'_k$ for $1 \leq k \leq |L|$. This is the case that one of the threads performs an action.

- $c \xrightarrow{a(l_i)} c'$ if for all $1 \leq i \leq |T|$, $c^i = c'^i$, $\exists 1 \leq j \leq |T|$ such that $l_j = \bot$ and $l'_j \in [1, |L|]$ and $l_k = l'_k$ for $k \neq j$. This is the case that thread $t_i$, acquires lock $l_j$.

- $c \xrightarrow{r(l_i)} c'$ if for all $1 \leq i \leq |T|$, $c^i = c'^i$, $\exists 1 \leq j \leq |T|$ such that $l_j \in [1, |L|]$ and $l'_j = \bot$ and $l_k = l'_k$ for $k \neq j$. This is the case that thread $t_i$, 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 $\mathcal{L}(l) = \{a(l), r(l)\}$ to be the set of lock operations on lock $l$ and $\mathcal{L} = \bigcup_{l \in \mathcal{L}} \mathcal{L}(l)$ to be the set of lock operations on all locks.

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

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

**Definition 2.** Let $\chi$ be the global execution of program $P$ leading to global configuration $c$. For thread $t_i$, lock $l_j$ such that $j \in \mathcal{S}(t_i, c)$, 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 $\chi$. Define acquisition history at configuration (state) $c$, $\mathcal{H}(t_i, c, \chi) = \bigcup_{l \in \mathcal{S}(t_i, c)} \mathcal{H}(t_i, l, c, \chi)$.

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 $\mathcal{H}(t_1, c_{o_1}, \chi_1) \sim \mathcal{H}(t_2, c_{o_2}, \chi_2)$, if $\exists l_1, l_2$ such that following conditions are true

- $l_1 \in \mathcal{S}(t_1, c_{o_1})$
• \( l_2 \in \mathcal{S}(t_2, c_{o_2}) \)

• \( l_1 \in \mathcal{H}(t_2, l_2, c_{o_2}, \chi_2) \)

• \( l_2 \in \mathcal{H}(t_1, l_2, c_{o_1}, \chi_1) \).

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 \( \text{EF}(o_1 \land o_2) \) 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 \( t_2 \), \( P \models \text{EF}(o_1 \land o_2) \) iff \( \exists \chi_a, \chi_b \) such that \( \chi_a \) leads \( t_1 \) to state \( o_1 \), \( \chi_b \) leads \( t_2 \) to state \( o_2 \) individually, and:

• \( \mathcal{S}(t_1, c_{o_1}) \cap \mathcal{S}(t_2, c_{o_2}) = \emptyset \)

• \( \mathcal{H}(t_1, c_{o_1}, \chi_1) \preceq \mathcal{H}(t_2, c_{o_2}, \chi_2) \)
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 tr(P) of program P as the start point, then tries to discover alternative schedulings of 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 \( \chi = x_1 x_2 \cdots x_{k+1} \), define execution trace \( tr \) corresponding to \( \chi \) as the sequence of events \( tr = e_1 e_2 \cdots e_k \) where \( x_1 \overset{e_1}{\rightarrow} x_2 \), 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

\[
\triangleright_i (A_i \cup \Sigma)^* \triangleleft_i
\]

\( \triangleright_i \) 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 \( e_1 \xleftrightarrow{d} e_2 \), 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 \( e_1 \nleftrightarrow e_2 \). Both \( \xleftrightarrow{d} \) and \( \nleftrightarrow \) 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 \( e_1 \nleftrightarrow e_2 \)
- \( \exists tr_3 \) 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 \( \not\exists k \) s.t. \( p < k < q, tr[k] = \triangleright_i \lor tr[k] = \triangleleft_i \), then \( \forall j \) such that \( p < j < q, tr[j] \in A_i \cup \mathcal{L} \).

**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 \( e_1 \xleftrightarrow{d} f \) and \( e_2 \xleftrightarrow{d} f \) 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

• $e_1 \leftrightarrow f$ and $e_2 \leftrightarrow f$

• $\exists e = tr_i[k_1], k_1 \leq k_4 < k_2$ 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 $\mathcal{S}$ represents for lock set and $\mathcal{A}$ represents for last acquisition:

```
PenelopeLockSetAndLastAcquisition(tr|t, I):
    A ← ∅
    S ← ∅
    for $w_i$ in $tr|t = w_1w_2 \cdots w_{|tr|t}|$:
        if $w_i = a(l)$ for some $l \in \mathcal{L}$:
            $S ← S \cup \{l\}$
            $A(l) ← i$
        if $w_i = r(l)$ for some $l \in \mathcal{L}$:
            $S ← S \setminus \{l\}$
        if $i \in I$:
            $A(i) ← A$
    return $S, A$
```

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 $O(|tr| \cdot |\mathcal{L}|)$ time with $O(|tr| \cdot |\mathcal{L}|)$ 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(V, T):
    P ← \{(ww, r), (rr, w), (rw, w), (wr, w), (ww, w)\}
    for $v$ in $V$:
        for $(E, F)$ in $P$:
            for $(t_i, t_j)$ s.t. $t_i, t_j \in T$ and $i \neq j$:
                for $e \in E|t_i, v, f \in F|t_j, v$:
                    if $S(t_i, e) \cap S(t_j, f) = \emptyset \land H(t_i, e) \not\sim H(t_j, f)$:
                        Atomicity violation found
```

9
In code above, $e, f$ are configurations when read/write actions happen. $\chi$’s are omitted from $\mathcal{H}$ since PENELope only observes one execution for any thread thus execution is not a variable.
Definition 9. A straight line grammar (SLG) is a special class of context free grammar (CFG). Formally, SLG is a tuple $(\Sigma, \Pi, P, S)$, where $\Sigma$ is the set of terminal symbols, $\Pi$ is the set of non-terminal symbols, $\Sigma \cap \Pi = \emptyset$, $P : \Pi \rightarrow (\Sigma \cup \Pi)^*$ is the production rule function and $S \in \Pi$ is the start symbol. For every $\pi \in \Pi$, $\pi \notin P(\pi)$. Define relation $\ni \in \Pi \times \Pi$, $\pi_1 \ni \pi_2$ if either of the following is true:

- $\pi_2 \in P(\pi_1)$
- $\exists \pi_3$ such that $\pi_1 \ni \pi_3$ and $\pi_3 \ni \pi_2$

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

Remark 1. $P$ 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 $s$, $P(s)$ 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, $w = w_1w_2 \cdots w_{\ell(w)}$. 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 $\pi$, the string
containing only terminal symbols produced following the production rules from \( \pi \) is denoted as \( w_{\pi \rightarrow} , \forall \pi \in \Pi , w_{\pi \rightarrow} \in \Sigma^* \). For terminal symbol \( \sigma \in \Sigma \), define \( w_{\sigma \rightarrow} = \sigma \).

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 \( \text{slg} \ g = (\Sigma, \Pi, P, S) \), height of symbol \( s \), denoted as \( h(s) \), is defined as follows (does not apply to start symbol \( S \)):

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

**Definition 12.** Given \( \text{slg} \ g = (\Sigma, \Pi, P, S) \), 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. \( \ell(w_{s \rightarrow}) \). The recursive definition of \( \ell \) is:

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

The string encoded by \( \text{slg} \) string can be reconstructed by doing invocation of \( \text{Derive}(S, P) \).

```
DERIVE(s, P):
    if s \in \Sigma:
        return [s]
    w_{s \rightarrow} \leftarrow \text{NIL}
    for i in 1 to |P(s)|, where P(s) = s_1 s_2 \cdots s_{|P(s)|}:
        w_{s \rightarrow} \leftarrow \text{concat}(w_{s \rightarrow}, \text{DERIVE}(s_i, P))
    return w_{s \rightarrow}
```
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 \( \Lambda : \mathcal{L} \times \Sigma^* \rightarrow N \), returning 1-based index of last lock acquisition operation \( a(l) \) in \( w \), or 0 if \( a(l) \notin w \).

\[
\Lambda(l, w) = \begin{cases} 
0 & \text{if } \nexists \ i \text{ such that } w_i = a(l) \\
\max\{i | 1 \leq i \leq |w| \land w_i = a(l)\} & \text{otherwise}
\end{cases}
\]

We can define last acquisition for string \( w \) compressed by symbol \( s \in \Sigma \cup \Pi \), i.e. \( w = w_{s \ldots} \), 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 \( \Lambda : \mathcal{L} \times (\Sigma \cup \Pi) \rightarrow N \), returning 1-based index of last acquisition of \( l \) in string \( w_{s \ldots} \), or 0 if \( a(l) \notin w_{s \ldots} \).

\[
\Lambda(l, s) = \begin{cases} 
1 & \text{if } s = a(l) \\
0 & \text{if } s \neq a(l) \land s \in \Sigma \\
\ell(s_1) + \Lambda(l, s_2) & \text{if } \mathcal{P}(s) = s_1s_2 \land \Lambda(l, s_2) 
eq 0 \\
\Lambda(l, s_1) & \text{if } \mathcal{P}(s) = s_1s_2 \land \Lambda(l, s_2) = 0
\end{cases}
\]
\[ \hat{\Lambda} : L \times (\Sigma \cup \Pi) \times \mathbb{N} \to \mathbb{N} \] can be defined by generalizing \( \Lambda \) and take an index as extra parameter. \( \hat{\Lambda}(l, s, \text{idx}) \) will return the last acquisition value of substring \( w_{s \odot [1, \text{idx}]} \). In particular, \( \Lambda(l, s) = \hat{\Lambda}(l, s, \ell(s)) \).

\[
\begin{aligned}
\hat{\Lambda}(l, s, \text{idx}) : \\
\text{if } \text{idx} = \ell(s) : \\
\quad \text{return } \Lambda(l, s) \\
\text{if } \hat{\Lambda}(l, s_2, \text{idx} - \ell(s_1)) > 0 : \\
\quad \text{return } \ell(s_1) + \hat{\Lambda}(l, s_2, \text{idx} - \ell(s_1)) \\
\text{else :} \\
\quad \text{return } \Lambda(l, s_1)
\end{aligned}
\]

**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 \( w = w_1 w_2 \cdots w_{|w|}, w_i \in \Sigma \), is a function \( \bar{\Delta} : L \times \Sigma^* \to \{-1, 0, 1\} \). \( \bar{\Delta}(l, a(l)) \) is defined to be 1, \( \bar{\Delta}(l, r(l)) \) is defined to be \(-1\) and \( \bar{\Delta}(l, a), a \in A \), is defined to be 0. \( \bar{\Delta}(l, w) = \sum_{i=1}^{|w|} \bar{\Delta}(l, w_i) \)

**Remark 2.** \( \bar{\Delta} : L \times \Sigma^* \to \{-1, 0, 1\} \) since there are only two possible state values, and \( tr|_t \) 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 \( tr|_t \) 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 \( \bar{\Delta}(l, tr|_t[i+1, j]) = 0 \), 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. \( w = w_{s \odot \ldots} \), 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 \( s \):
\[ \Delta(l, s) = \begin{cases} 
1 & s = a(l) \\
-1 & s = r(l) \\
0 & s \not\in \mathcal{L}(l) \land s \in \Sigma \\
\Delta(l, s_1) + \Delta(l, s_2) & \mathcal{P}(s) = s_1s_2 
\end{cases} \]

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

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

For the rest of this thesis, for a given lock \( l \), we use \( \Lambda_l \) to denote both \( \Lambda \) and \( \hat{\Lambda} \), use \( \Delta_l \) to denote both \( \Delta \) 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 \( \mathcal{G} \) and \( \mathcal{A} \) at the specific index of trace.
5.2 Computing Indices for Events of Interest

We need to track four sets of events of interest: \( \mathbf{r}, \mathbf{w}, \mathbf{ww}, \mathbf{aa} \) where no set should contain two events with the same \((S, A)\) pair. First we have the following observation:
Observation 1. \((S, A)\) 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 \(O(|t|)\) time with \(O(1)\) additional space are shown below.

Function \texttt{ReadIndices} takes projection of uncompressed trace to a thread \(t\), and outputs the index set of read events.

\[
\texttt{ReadIndices}(tr):
\begin{align*}
\mathcal{I} & \leftarrow \text{NIL} \\
b & \leftarrow \top \\
\text{for } i \text{ in } 1 \text{ to } |tr|, \text{ where } tr = w_1w_2\cdots w_{|tr|}: \\
\text{if } w_i = r(v) \text{ for some } v \in V \text{ and } b = \top: \\
\quad \mathcal{I} & \leftarrow \mathcal{I} \text{ snoc}(v, i) \\
\quad b & \leftarrow \bot \\
\text{if } w_i = \triangleright \text{ or } w_i \in L:\ \\
\quad b & \leftarrow \top \\
\text{return } \mathcal{I}
\end{align*}
\]

Function \texttt{WriteIndices} computing indices of write events can be obtained by modifying the if condition in \texttt{ReadIndices} from

\[w_i = r(v)\]

to

\[w_i = w(v)\]

Function \texttt{WriteWriteIndices} computes indices of events sandwiched between two write events.
**WriteWriteIndices**

\[
\begin{align*}
I & \leftarrow \text{NIL} \\
\text{PREV} & \leftarrow \text{NIL} \\
b & \leftarrow \top \\
\text{for } i \text{ in } 1 \text{ to } |\text{tr}|, \text{ where } \text{tr} = w_1 w_2 \cdots w_{|\text{tr}|}: \\
\quad \text{if } w_i = W(v) \text{ for some } v \in V: \\
\qquad \text{if } \text{PREV} \neq \text{NIL}: \\
\qquad \quad I \leftarrow I \text{snoc} \text{PREV} \\
\qquad \quad \text{PREV} \leftarrow \text{NIL} \\
\qquad \text{if } b = \top: \\
\qquad \quad \text{PREV} \leftarrow (v, i) \\
\qquad \quad b \leftarrow \bot \\
\quad \text{if } w_i = \triangleright \text{ or } w_i \in L: \\
\qquad \quad b \leftarrow \top \\
\quad \text{if } w_i = \triangleleft: \\
\quad \quad \text{PREV} \leftarrow \text{NIL} \\
\text{return } I
\end{align*}
\]

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

\[
w_i = W(v)
\]

to

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

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

**Remark 3.** In \( I|_t \) constructed above, no two events of the same kind \( e,e' \) have identical \((S,A)\) pair but it is possible for \( e,e' \) to have identical \((S,H)\) 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: $\mathcal{M}_\mathcal{S}$ maps event indices to the corresponding lock sets and $\mathcal{M}_A$ maps event indices to corresponding last acquisitions.

\begin{algorithm}
\textbf{LOCKSETSANDLASTACQUISITION}(\text{ctr}, \text{I}) :
  \begin{algorithmic}
  \State $\zeta(l) \leftarrow 0$
  \State $A(l) \leftarrow 0$
  \State $c \leftarrow 0$
  \State $\mathcal{M}_\mathcal{S} \leftarrow \emptyset$
  \State $\mathcal{M}_A \leftarrow \emptyset$
  \For{$i$ in 1 to $|\text{ctr}|$, where $\text{ctr} = w_1 w_2 \cdots w_{|\text{ctr}|}$}:
    \While{$\text{car}(\text{I}) < c + \ell(w_i)$}:
      \State $idx \leftarrow \text{car}(\text{I}) - c$
      \State $A' \leftarrow A$
      \For{$l$ in $\mathcal{L}$}:
        \State $\zeta'(l) \leftarrow \zeta(l) + \Delta_l(w_i, idx)$
        \If{$\Lambda_l(w_i, idx) > 0$}:
          \State $A'(l) \leftarrow c + \Lambda_l(w_i, idx)$
          \State $\mathcal{M}_\mathcal{S}(\text{car}(\text{I})) \leftarrow \{l | \zeta'(l) = 1\}$
          \State $\mathcal{M}_A(\text{car}(\text{I})) \leftarrow \text{sort}(A')$
          \State $\text{I} \leftarrow \text{cdr}(\text{I})$
        \EndIf
      \EndFor
    \EndWhile
    \For{$l$ in $\mathcal{L}$}:
      \State $\zeta(l) \leftarrow \zeta(l) + \Delta_l(w_i)$
      \If{$\Lambda_l(w_i) > 0$}:
        \State $A(l) \leftarrow c + \Lambda_l(w_i)$
      \EndIf
      \State $c \leftarrow c + \ell(w_i)$
  \EndFor
  \State \Return $\mathcal{M}_\mathcal{S}, \mathcal{M}_A$
\end{algorithmic}
\end{algorithm}

Remark 4. Notice $A$ 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 $O(|\text{ctr}| + |\mathcal{L}| \cdot |\text{I}| \cdot h_{\text{max}})$ time with $O(|\mathcal{L}| \cdot |\Pi|)$ space, where $h_{\text{max}} = \max_{\pi \in \Pi} h(\pi)$. 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 $A_1, A_2$ and lock sets $S_1, S_2$, we can use algorithm below to determine if corresponding acquisition histories are compatible in $O(|L|)$ time with $O(1)$ additional space.

<table>
<thead>
<tr>
<th>IsCompatible($S_1, A_1, S_2, A_2$):</th>
</tr>
</thead>
<tbody>
<tr>
<td>while car($A_1$) not in $S_1$:</td>
</tr>
<tr>
<td>$A_1 \leftarrow$ cdr($A_1$)</td>
</tr>
<tr>
<td>while car($A_2$) not in $S_2$:</td>
</tr>
<tr>
<td>$A_2 \leftarrow$ cdr($A_2$)</td>
</tr>
<tr>
<td>$A_1 \leftarrow$ filter($l \in S_1 \lor l \in S_2, A_1$)</td>
</tr>
<tr>
<td>$A_2 \leftarrow$ filter($l \in S_1 \lor l \in S_2, A_2$)</td>
</tr>
<tr>
<td>while $A_1 \neq$ NIL and $A_2 \neq$ NIL:</td>
</tr>
<tr>
<td>if car($A_1$) $\neq$ car($A_2$):</td>
</tr>
<tr>
<td>return false</td>
</tr>
<tr>
<td>$A_1 \leftarrow$ cdr($A_1$)</td>
</tr>
<tr>
<td>$A_2 \leftarrow$ cdr($A_2$)</td>
</tr>
<tr>
<td>return true</td>
</tr>
</tbody>
</table>
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
\( \mathcal{D} : \Sigma \cup \Pi \times \Sigma \cup \Pi \to \Pi \times \mathbb{N} \) is the digram map. \( \mathcal{D} \) maps a pair of symbols to a nonterminal symbol and a occurrence count.

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

\[\text{SEQUITURCOMPRESS}(w):\]
\[
\mathcal{P}(S) \leftarrow [w_1]
\]
\[\text{for } i \text{ in } 2 \text{ to } |w|, \text{ where } w = w_1w_2 \cdots w_{|w|}:\]
\[
\mathcal{P}(S) \text{ snoc } s \leftarrow \mathcal{P}(S)
\]
\[n \leftarrow w_i
\]
\[\text{while } (s, n) \text{ in } \mathcal{D}:\]
\[
(\pi, c) \leftarrow \mathcal{D}(s, w_i)
\]
\[\text{if } c = 1:\]
\[
\varpi \leftarrow \text{NEWSYMBOL()}
\]
\[
\text{DIGRAMUNIQUENESS}(\pi, (s, n), \varpi)
\]
\[
\mathcal{D}(s, n) \leftarrow (\varpi, 2)
\]
\[
\mathcal{P}(S) \leftarrow \mathcal{P}(S) \text{ snoc } \varpi
\]
\[\text{else}:\]
\[
\mathcal{D}(s, n) \leftarrow (\pi, c + 1)
\]
\[
\mathcal{P}(S) \leftarrow \mathcal{P}(S) \text{ snoc } \pi
\]
\[\text{if } |\mathcal{P}(S)| > 1:\]
\[
\mathcal{P}(S) \text{ snoc } s \text{ snoc } n \leftarrow \mathcal{P}(S)
\]
\[\text{else}:\]
\[
\text{continue to next for loop iteration}
\]
\[
\mathcal{D}(s, n) \leftarrow (S, 1)
\]
\[
\mathcal{P}(S) \leftarrow \mathcal{P}(S) \text{ snoc } s \text{ snoc } n
\]
\[\text{return } \mathcal{P}
\]

For \texttt{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 PENE-LOPE. 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>
<thead>
<tr>
<th>Application (LOC)</th>
<th>Input</th>
<th>Threads</th>
<th>Variables</th>
<th>Locks</th>
<th>Trace Length</th>
<th>Compressed Trace Length</th>
<th>Nonterminals</th>
<th>Compression Rate</th>
<th>Time (Uncompressed)</th>
<th>Time (Compressed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector (1.3K)</td>
<td>VectorTest</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>142</td>
<td>83</td>
<td>29</td>
<td>59%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>VectorTest1</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>231</td>
<td>122</td>
<td>53</td>
<td>53%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>VectorTest2</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>248</td>
<td>131</td>
<td>44</td>
<td>53%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>VectorTest3</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>231</td>
<td>95</td>
<td>46</td>
<td>41%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>VectorTest4</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>159</td>
<td>86</td>
<td>33</td>
<td>54%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>Stack (1.4K)</td>
<td>StackTest</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>136</td>
<td>90</td>
<td>10</td>
<td>66%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>StackTest1</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>209</td>
<td>125</td>
<td>38</td>
<td>60%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>StackTest2</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>231</td>
<td>132</td>
<td>50</td>
<td>57%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>StackTest3</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>265</td>
<td>140</td>
<td>41</td>
<td>53%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td></td>
<td>StackTest4</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>127</td>
<td>77</td>
<td>13</td>
<td>61%</td>
<td>0.01s</td>
<td>0.01s</td>
</tr>
<tr>
<td>elevator (566)</td>
<td>data</td>
<td>3</td>
<td>65</td>
<td>8</td>
<td>26915</td>
<td>8670</td>
<td>1179</td>
<td>31%</td>
<td>10.71s</td>
<td>8.71s</td>
</tr>
<tr>
<td></td>
<td>data2</td>
<td>5</td>
<td>113</td>
<td>8</td>
<td>54566</td>
<td>13359</td>
<td>2511</td>
<td>24%</td>
<td>7.12s</td>
<td>3.64s</td>
</tr>
<tr>
<td></td>
<td>data3</td>
<td>5</td>
<td>457</td>
<td>50</td>
<td>329913</td>
<td>69757</td>
<td>18089</td>
<td>21%</td>
<td>12m49s</td>
<td>3m18s</td>
</tr>
<tr>
<td>tsp (794)</td>
<td>map13</td>
<td>2</td>
<td>588</td>
<td>2</td>
<td>32250186</td>
<td>6116075</td>
<td>712365</td>
<td>19%</td>
<td>4m40s</td>
<td>1m40s</td>
</tr>
<tr>
<td></td>
<td>map14</td>
<td>4</td>
<td>652</td>
<td>2</td>
<td>14026944</td>
<td>3132490</td>
<td>489379</td>
<td>22%</td>
<td>3m21s</td>
<td>42s</td>
</tr>
</tbody>
</table>
In this paper, we presented an improved algorithm detecting atomicity violation based on \textsc{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 \textsc{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


