Localization of Linearizability Faults on the Coarse-grained Level

Zhenya Zhang, Peng Wu, Yu Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences

***Abstract*—Linearizability is an important correctness criterion that guarantees the safety of concurrent data structures. Due to the nondeterminism of concurrent executions, reproduction and localization of a linearizability fault still remain challenging. The existing work mainly focuses on model checking the thread schedule space of a concurrent program on a fine-grained (state) level, and hence suffers from the severe problem of state space explosion. This paper presents a tool called CGVT to build a small test case that is sufficient enough for reproducing a linearizability fault. Given a possibly long history that has been detected non-linearizable, CGVT first locates the operations causing a linearizability violation, and then synthesizes a short test case for further investigation. Moreover, we present several optimization techniques to improve the effectiveness and efficiency of CGVT. We have applied CGVT to 10 concurrent objects, while the linearizability of some of the concurrent objects is unknown yet. The experiments show that CGVT is powerful and efficient enough to build the test cases adaptable for a fine-grained analysis.**

I. Introduction

Linearizability [1] is a widely accepted correctness criterion that guarantees the safety of concurrent data structures or concurrent objects. Intuitively, a concurrent history of a shared object is linearizable if each operation of the object appears to take effect instantaneously at some point, called linearization point, between the invocation and response of the operation, and the history can be serialized as a sequence of the operations that is consistent with the sequential specification of the object. Linearizability checking is still a challenging task for concurrent objects.

Due to the nondeterminism of concurrent executions, reproduction and localization of a linearizability fault is notoriously difficult. This problem can be addressed by systematically exploiting all possible fine-grained traces that are composed of memory access instructions [2], [3], which however suffers from the severe problem of state space explosion. Although techniques such as iterative context bounding [4] can help improve the scalability of this systematic testing approach, it would work better with small test cases. Besides, such test cases can be generated through static analysis or empirical evidence [5], [6], but lack accuracy, especially for very complicated objects or concurrency faults. ConTeGe [7] can fully automatically produce a large number of small-scale traces, which are composed of operations with random arguments, but may not manifest potential faults. VeriTrace [8] can produce traces that can trigger various buggy executions, but these traces may contain redundant operations, therefore raise the complexity of further analysis unnecessarily.

In this paper, we present a tool, CGVT, to build small test cases that are sufficient enough for triggering linearizability faults. Given a possibly long history that has been detected non-linearizable, CGVT first locates the operations causing a linearizability violation. Note that the operations resulting in a non-linearizable execution are usually not located as would be expected intuitively.

Example 1 Fig. 1 shows a simplified PairSnapShot [9], where an array `d` (of size 2) is shared between two concurrent threads. A `write(i,v)` operation writes `v` to `d[i]`, while a `read` operation reads the values of `d[0]` and `d[1]` together as a pair. A correctness criterion is that `read` should always return the values of the same moment. However, Fig. 2 illustrates a concurrent execution in which the return values of `read` do not exist at any moment of the execution. The 'x' labels in Fig. 2 mark the moments when each instruction takes effect. Then, this concurrent execution with the 5 operations explains exactly the cause of the linearizability violation on a fine-grained state level.

```
PairSnapShot:              Pair read(){
                             while(true){
  int d[2];                    int x = d[0];    #2
                               int y = d[1];    #3
  write(i,v){                  if(x == d[0])    #4
    d[i] = v;  #1                return (x,y);
  }                          }
                           }
```

Fig. 1: Source code of PairSnapShot

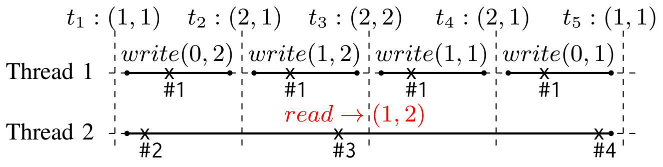


Fig. 2: A buggy trace of PairSnapShot

Example 2 Fig. 3 presents a linked-list based set [10]. Fig. 4 shows a non-linearizable trace of the set, where O_1 and O_2 appear to add 8 and 9, respectively, to the list, but later O_3 does not find 8 in the list. Herein, the synchronized block #3 can be treated as an atomic instruction. The actual cause

DOI reference number: 10.18293/SEKE2017-145

of the linearizability violation is as follows: after both add operations locate the same nodes pred and curr, add(8) first sets pred.next to node 8, then add(9) sets it to node 9, which makes node 8 removed from the list. It can be seen that although it is the wrong return value of contain(8) that reveals the linearizability violation, the root cause of this non-linearizable trace only lies in the concurrent add operations.

```
Set:
    add(int key){
        Node pred,curr = locate(key);  #1
        if(curr.key == key)            #2
            return false;
        synchronized(){                #3
            Node n = new Node(key);
            n.next = curr;
            pred.next = n;
            return true;
        }
    }
```

Fig. 3: Source code of a buggy set

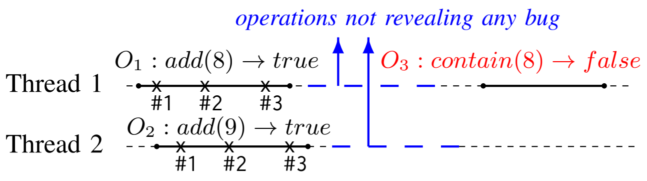


Fig. 4: A buggy concurrent trace of Set

Example 1 shows a scenario where a violation is "revealed" immediately after the corresponding linearizability fault is "triggered", while Example 2 shows another scenario where a linearizability fault is "triggered" possibly long before the corresponding violation is "revealed". In this work, we aim to localize, from a given non-linearizable trace, the operations that essentially cause the linearizability violation and then synthesize a small test case that can trigger the same linearizability fault.

Contributions. The main contributions of this paper are three-fold.

- We make clear the relationship between "triggering" and "revealing" linearizability faults, and formally define *minimum test cases* that are sufficient to trigger linearizability faults;
- We propose the framework of CGVT for building minimum test cases, as well as a D-PCT based optimization and a heuristic rule based acceleration technique;
- We implement a prototype tool, CGVT, which was experimented with 10 concurrent objects. The experiment results show that the tool is effective and efficient enough to build the test cases adaptable for a fine-grained analysis.

Related work. The existing work on linearizability checking mainly aims to solve the state space explosion problem to accelerate the checking of concurrent histories. Automated linearizability checking algorithms [11], [12] suffer from a performance bottleneck. Based on [12], optimized algorithms were proposed through partial order reduction [13] or compositional reasoning [6]. Model checking was applied for linearizability checking, with simplified first-order formulas that can help improve efficiency [5], [14]. Fine-grained traces were introduced in [8] to accelerate linearizability checking.

The output of our work is a small test case that is adaptable for concurrency fault analysis on a fine-grained level, such as [2]–[4]. A framework with interleaving test generation heuristics was introduced in [2] for bug detection and replay. A constraint-based symbolic analysis method was proposed in [3] to diagnose concurrency bugs. An acceleration technique was presented in [4] for model checking thread schedule spaces. Unlike these bug reproduction and localization techniques [2], [4], [7] that aim at general concurrency bugs, our work aims at linearzability violation specifically.

Our work can also facilitate debugging concurrency bugs. Recent work on the debugging of concurrency bugs often applies the techniques for sequential programs to concurrent programs, such as assertions [15], or breakpoints [16].

Organization. The rest of the paper is organized as follows. Section II introduces the trace model and the formal definition of minimum test cases. Section III presents the implementation of CGVT, with the D-PCT based optimization and the heuristic rule based acceleration technique. Section IV discusses the results of experiments. Section V concludes the paper.

II. Trace Model

A. Linearizability

Let $\mathbb{M}, \mathbb{T}, \mathbb{O}, \mathbb{V}$ respectively denote the set of operation names, thread identifiers, operation identifiers and values. Then, $C = \{m(v_a)_o^t : m \in \mathbb{M}, v_a \in \mathbb{V}, o \in \mathbb{O}, t \in \mathbb{T}\}$ and $R = \{m_o^t \to v_r : m \in \mathbb{M}, v_r \in \mathbb{V}, o \in \mathbb{O}, t \in \mathbb{T}\}$ represent the set of operation *invocation* events and *response* events, respectively. We denote the operation identifier of an event $e \in (C \cup R)$ as $\mathtt{op}(e)$. Events $c \in C$ and $r \in R$ match each other, denoted $c \diamond r$, if $\mathtt{op}(c) = \mathtt{op}(r)$. A pair of matching events forms an operation O, written as $m(v_a) \to v_r$.

A sequence $S = e_1 e_2 \cdots e_n \in (C \cup R)^*$ is *well-formed* if:

- Each response is preceded by a matching invocation: $e_j \in R$ implies $e_i \diamond e_j$ for some $1 \leq i < j \leq n$
- Each operation identifier is used in at most one invocation/response: $\mathtt{op}(e_i) = \mathtt{op}(e_j)$ and $1 \leq i < j \leq n$ implies $e_i \diamond e_j$

A well-formed sequence S can be treated as a partial order set $(H, \prec_H)$, where H is called a history (or trace) composed of the operations formed by matching events in S, and $\prec_H$ is the *happen-before* relation in S. For operations $O_1, O_2 \in H$, $O_1 \prec_H O_2$ if the response of O_1 is ahead of the invocation of O_2 in S. Let $\mathtt{size}(H)$ denote the *size* of H, i.e., the number of the operations involved in history H.

An invocation event is *pending* in H if no matching response event follows it. An *extension* of H, denoted $\mathcal{E}(H)$, is a history constructed by appending response events to all the pending invocation events in H. Sometimes we ignore these pending invocation events and get $\mathcal{C}(H)$, the subsequence of H consisting of all the matching invocation and response events in H.

If H is a total order set, then H is *sequential*. A *specification* of an object is the set of all the sequential histories that satisfy the correctness criterion of the object.

Definition 1 (Linearizability)**.** *A history H of a concurrent object is linearizable if there is an extension $\mathcal{E}(H)$ and a history S in the specification of the object such that:*

- *Elements of $\mathcal{E}(H)$ and S are same;*
- *$\prec_H \subseteq \prec_S$, i.e., if $O_1 \prec_H O_2$, then $O_1 \prec_S O_2$.*

Here, S is called a witness of H.

B. Minimum test case

Operations with the same thread identifier form a sequential program P_s, an execution of which results in a sequential history. A concurrent program P_c with n threads is a set of n sequential programs. An execution of P_c is a concurrent execution of the n sequential programs, each starting at an arbitrary moment.

A *state* of an object at some moment is a mapping from the shared variables to their values. Considering the initial state of an object as default, we can use a sequential program P_s to represent a state, since P_s can determine it definitely. For concurrent program P_c, thread schedules decide what states are reached. A *schedule* for P_c is a sequence of memory access instructions in P_c.

A *test case* is defined as a triple $\mathtt{t_c} = (P_s, P_c, P_r)$, where P_s is a state, P_c is a concurrent program and P_r is a sequential program. Starting from state P_s, an execution of $\mathtt{t_c}$ runs the sequential programs in P_c concurrently to "trigger" a bug if any, and then runs P_r at last to "reveal" the bug. The trace of an execution of $\mathtt{t_c}$ depends on the schedule of the execution for P_c. If there exists a schedule leading a trace of $\mathtt{t_c}$ non-linearizable, we say that $\mathtt{t_c}$ is *potential to trigger a linearizability fault* or *buggy*; otherwise $\mathtt{t_c}$ is *correct*.

Definition 2 (minimum test case)**.** *A test case $\mathtt{t_c}$ is an minimum test case if it is potential to trigger a linearizability fault and removal of any operation in its P_c results in a correct test case.*

Examples The trace shown in Fig. 2 results from a minimum test case where $P_s = \{write(0,1), write(1,1)\}$, $P_c = \{\{write(0,2), write(1,2), write(1,1), write(0,1)\}, \{read\}\}$, and $P_r = \emptyset$. The trace shown in Fig. 4 results from a minimum test case where $P_s = \{add(7), add(10)\}$, $P_c = \{\{O_1\}, \{O_2\}\}$, and P_r is a sequential program ending with O_3.

III. CGVT

In this section, we present the basic implementation of CGVT, a D-PCT based optimization and a heuristic rule based acceleration technique. CGVT is divided into two phases, detection and localization. The former is to acquire a possibly long trace that is non-linearizable, and the latter is to locate the operations causing a linearizability fault, and synthesize a minimum test case for it.

A. Basic implementation

Detection phase. VeriTrace [8] is an off-line tool for automated linearizability checking. Given a concurrent object as an input, VeriTrace automatically runs concurrently a couple of threads, each executing a certain number of operations, and records the history H of the concurrent execution on the object. Then, WGL [13], a linearizability checking algorithm, enumerates possible sequential histories to search for a witness of the concurrent history H. If no witness of H is found, then H is detected non-linearizable and will be analyzed in the next phase for localization.

Localization phase. We define a *prefix* of the concurrent history H, denoted $H_p(n)$, as a sub-history of H composed of the former n events in H.

Theorem 1. *H is linearizable if and only if every prefix of H is linearizable.*

Proof For the "only if" direction, suppose that H is linearizable and there exists a non-linearizable prefix $H_p(k)$. Then, any extension $\mathcal{E}(H_p(k))$ is non-linearizable, which implies that the addition of operations $O' \in \mathcal{E}(H_p(k)) \setminus \mathcal{C}(H_p(k))$ can not serialize it. Neither can the addition of operations O'' such that $\forall m(m \in \mathcal{C}(H_p(k)) \wedge m \prec_H O'')$, because such O'' can only be ordered after the operations of $\mathcal{C}(H_p(k))$ in a witness. Therefore, there is no witness for any superset of $\mathcal{C}(H_p(k))$, which implies H is non-linearizable, a contradiction.

For the "if" direction, if every prefix of H is linearizable, it is obvious that H is linearizable. □

Algorithm 1 Bug Localization

```
$OpOrder$. SORTBYSIZE()
$s_o \leftarrow |OpOrder|$
function LOCALIZE
    for $i \leftarrow \{s_o, \cdots, 0\}$ do
        $P_s \leftarrow OpOrder[i]$
        $P_c \leftarrow H_p(b) \setminus P_s$
        $\mathtt{t_c} \leftarrow$ NEW TESTCASE($P_s, P_c, \varnothing$)
        if !CHECK($\mathtt{t_c}$) then break
        end if
    end for
    for $j \leftarrow \{2, \cdots, \mathtt{size}(\mathtt{t_c}.P_c) - 2\}$ do
        $P_c \leftarrow \mathtt{t_c}.P_c[0:j]$
        $P_r \leftarrow OpOrder[s_o] \setminus \mathtt{t_c}.P_s \setminus P_c$
        $\mathtt{t_c} \leftarrow$ NEW TESTCASE($\mathtt{t_c}.P_s, P_c, P_r$)
        if !CHECK($\mathtt{t_c}$) then return $\mathtt{t_c}$
        end if
    end for
end function

function CHECK($\mathtt{t_c}$)
    while $k \leftarrow \{0, \cdots, \sigma\}$ do
        if !$L_n$($\mathtt{t_c}$.EXECUTE()) then return $false$
        end if
    end while
    return $true$
end function
```

From Theorem 1, it can be seen that for the given non-linearizable history H, there exists a non-linearizable prefix $H_p(b)$ whose size is smaller than any other non-linearizable prefixes of H. It is obvious that $H_p(b)$ involves the operations triggering the linearizability fault, since otherwise $H_p(b)$ would be linearizable. We can use $H_p(b)$ to construct a minimum test case for a fine-grained bug localization algorithm.

Algorithm 1 presents the algorithm for the coarse-grained bug localization. $OpOrder$ is a set of witnesses of all prefixes of $H_p(b)$. We sort the witnesses in $OpOrder$ by their sizes (line 1). The process of building a minimum test case contains two steps:

1) Select a prefix of $H_p(b)$, serialize it as P_s (lines 4-5), and then set the remaining suffix of $H_p(b)$ to be P_c (line 6), check the test cases iteratively until a buggy one is found (lines 8-9);
2) Set a new P_c to be a prefix of the previous P_c (line 12), then serialize the remaining operations as P_r (line 13), and check the test cases iteratively until a buggy one is found (lines 15-16).

Note that the serialization in lines 5 and 13 is directly based on the sequential traces in $OpOrder$, and in practice, we can get both $H_p(b)$ and $OpOrder$ easily through the "cache" mechanism of the WGL algorithm in the detection phase.

Function CHECK (lines 20-26) is used for checking whether a test case is buggy, where L_n is a predicate judging the linearizability of a trace. Here, σ is a threshold for the number of times the test case is executed. Our experiments show that using the setting of σ in the detection phase is always enough to detect the fault in the localization phase.

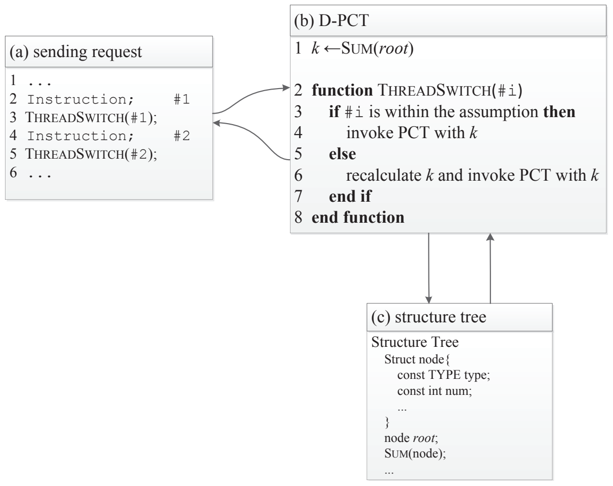
B. D-PCT based implementation

D-PCT based detection. The concurrent executions induced by VeriTrace closely depend on the run-time environment. Theoretically, for a concurrent program that contains n threads, each executing at most k instructions, the total number of its schedules is $\frac{(nk)!}{(k!)^n} \geq (n!)^k$, exponentially dependent on n and k. It is difficult to detect the faults that only appear under specific (complicated) schedules.

In order to address this problem, we adapt PCT [17], a scheduling algorithm with a much higher probabilistic guarantee of finding bugs, as D-PCT (Dynamic PCT) and apply it to CGVT, as shown in Fig. 5.

- In Fig. 5(a), during a concurrent execution in the detection phase, THREADSWITCH requests are called to ask whether a thread switch should take place currently;
- In Fig. 5(b), D-PCT responds immediately by invoking the PCT algorithm with the number of instructions (k).

Originally in PCT, k is required no less than the number of the instructions so that PCT can generate a schedule covering all the instructions, while in CGVT, k can not be pre-specified before the execution. Here, we model the instructions as a *structure tree*, by which we calculate k dynamically, as shown in Fig. 5(c).


(a) sending request
```
...
Instruction;       #1
THREADSWITCH(#1);
Instruction;       #2
THREADSWITCH(#2);
...
```
(b) D-PCT
```
k ← SUM(root)

function THREADSWITCH(#i)
    if #i is within the assumption then
        invoke PCT with k
    else
        recalculate k and invoke PCT with k
    end if
end function
```
(c) structure tree
```
Structure Tree
  Struct node{
    const TYPE type;
    const int num;
    ...
  }
  node root;
  SUM(node);
  ...
```
Fig. 5: Application of D-PCT

We split the instructions of an operation into different units: *sequential units* contain instructions executed sequentially such as #1 in Fig. 1; *loop units* contain instructions executed iteratively such as #2,#3,#4 in Fig. 1; *selection units* contain several *branches*, each containing instructions executed conditionally such as #3 in Fig. 3. Moreover, each *loop unit* and each *branch* can further be split until all the instructions belong to a *sequential unit*. The instructions in an operation can be built as the following *structure tree*:

1) The root represents all the instructions of the operation;
2) Each child node of the root represents one of the three units, from left to right following the program order of the instructions;
3) Each child node of a *selection unit* node is one of its *branches*;
4) The child nodes of a *loop unit* or *branch* node represents its lower-level units until *sequential units* as leaves.
5) Each node holds two constants: `type` and `num`. `type` is one of `R`, `L`, `C`, `B`, `S` respectively representing root, *loop unit, selection unit, branch, sequential unit*. For a node of type `R`, `L`, `C` or `B`, `num` is the number of its child nodes; while for a node of type `S`, `num` is the number of the instructions in the *sequential unit*.

Example Fig. 6 presents the structure trees of the three operations in Fig. 1 and Fig. 3.

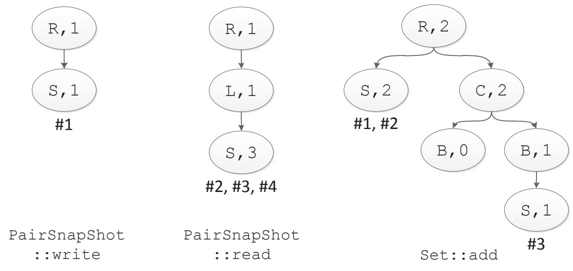


Fig. 6: Structure tree

Algorithm 2 Calculation of k

```
1:  function SUM($node$)
2:      $value \leftarrow 0$
3:      switch $node$.type
4:          case (S):                                        ▷ sequential unit
5:              $value \leftarrow value + node$.num
6:              break
7:          case (R|L|B):                          ▷ root, loop unit, branch
8:              for $n \in node.child$ do
9:                  $value \leftarrow value + \theta *$SUM(n)
10:             end for
11:             break
12:         case (C):                                        ▷ selection unit
13:             for $n \in node.child$ do
14:                 $value \leftarrow value + max($SUM(n) )
15:             end for
16:             break
17:     end switch
18:     return $value$
19: end function
```

With the help of a structure tree, D-PCT can make decision on thread switching as shown in Fig. 5(b). At first, D-PCT calculates k tentatively through function SUM (line 1), assuming that loop units iterate θ times and selection units select the branch with the maximum number of instructions. Then, each time a THREADSWITCH request is called, D-PCT checks whether the instruction, after which the request is called, is beyond the assumption (line 3). If not, D-PCT responds by invoking PCT directly (line 4); otherwise, D-PCT recalculates the currently remaining k in a way similar to line 1 and then invokes PCT (line 6).

Algorithm 2 presents the concrete algorithm of function SUM. It recursively searches the structure tree and adds up the number of the instructions of different units assuming that loop units iterate θ times (line 9) and selection units select the branch with the maximum number of instructions (line 14). Here, for a node of type L, if its number of iterations is known, θ is set accordingly, otherwise $\theta = 1$; for a node of type R or B, $\theta = 1$.

D-PCT based localization. In addition to Fig. 5(b), D-PCT records the instruction attached within each request to form the actual schedules during the detection phase. Then in the localization phase, these schedules are applied to function CHECK() in Algorithm 1 to guide the executions of P_c. In this way, the faults triggered under these schedules will be reproduced in the localization phase.

C. A heuristic rule based acceleration

We present a heuristic rule to make CGVT perform better. The rule is based on an **observation** that in P_c of a minimum test case (P_s, P_c, P_r), there exists a thread in which only one operation is executed.

This observation comes from the minimum test cases we have collected, which are presented in Table I. In Column 2, we call a test case is of type 1 if the test case "reveals" a fault immediately after the fault is"triggered", like the test case for PairSnapShot shown in Fig. 2; while a test case is of type 2 if the test case "reveals" a fault possibly long after the fault is "triggered", like the test case for the linked-list based set shown in Fig. 4. Column 3 reports the number of the operations needed in the concurrent threads of P_c in each minimum test case. Hence, we conjecture that whichever type a test case belongs to, one operation in some thread may be sufficient to trigger a fault. This observation is useful for acceleration if test cases built accordingly can still trigger faults.

TABLE I: Operation quantity on threads

	Type	OP Quantity
PairSnapShot [9]	1	1, 4
SimpleList [10]	2	1, 1
SimpleList (Size) [8]	1	1, 2
LockFreeList [18]	1	1, 1
K-Stack [19]	2	1, 1
OptimisticQueue [20]	1	1, 1
Snark [21]	1	1, 2
TreiberStack [5]	2	1, 3

IV. EXPERIMENT AND EVALUATION

Table II shows the experiment on 10 concurrent objects. Some of the objects are from the previous work and known to us, while others whose names begin with "BU" are adapted versions of the existing objects and unknown to us. The specification for the first 6 objects is *Set*. The specification for "OPQueue" and "BUQueue" is *FIFO-Queue*. The specification for the last two objects is *Stack*.

We compare the 3 versions of CGVT, respectively the basic version (B) in Section III-A, the D-PCT based version (D) in Section III-B, and the heuristic rule based version (H) in Section III-C. In the detection phase, for each object, n_s (possibly long) histories (Col. 4-6) are examined, each composed of $n_t * n_o$ (Col. 3) operations with n_t the number of the threads and n_o the number of the operations executed by each thread. Among these histories, n_l histories are detected non-linearizable (Col. 4-6). Note that the occurrence of a non-linearizable history is the prerequisite for localization, and the proportion reported in Col. 4-6 reflects the difficulty in detecting the linearizability faults. The average time costs for linearizability checking of these histories are listed in Col. 7-9. In the localization phase, selecting a non-linearizable history from the detection phase, CGVT works out minimum test cases of sizes shown in Col. 10-12. The time costs of localization are listed in Col. 13-15, mainly for the two calls of CHECK. The tool and benchmarks are available at *https://github.com/choshina/CGVT*.

We have the following observations from Table II:

1) The concurrent objects that interest us most are BUList3 and CGListS. For these two objects, version B cannot detect the bugs, but version D can. This shows that D-PCT enhances the CGVT's ability in detecting certain complicated bugs.
2) Comparing the time cost of version H with that of version D, we conclude that version H improves the efficiency.

TABLE II: Evaluation of CGVT

			Detection phase						Localization phase					
Object	Operations	Hist. size ($n_t * n_o$)	$!L_n$/Total (n_l/n_s)			Aver. time (ms)			$\mathtt{t_c}.P_c$ size			Time cost (ms)		
			B	D	H	B	D	H	B	D	H	B	D	H
LFList	add;remove	2*500	5/20	2/20	8/20	643	611	237	2	2	2	154	233	207
CGList	add;contain	2*50	20/20	20/20	20/20	33	24	23	2	2	2	51	256	219
CGList	add;remove;size	2*500	0/80	3/80	1/80	660	682	483	-	4	3	-	756	832
BUList1	add;remove	2*500	7/20	5/20	5/20	488	704	532	2	2	2	162	245	311
BUList2	add;remove	2*50	18/20	20/20	20/20	31	35	28	2	2	2	83	365	247
BUList3	add;remove	2*500	0/80	2/80	3/80	625	603	524	-	2	2	-	326	335
OPQueue	enqueue;dequeue	2*500	4/20	19/20	16/20	2430	1412	832	2	3	2	378	796	683
BUQueue	enqueue;dequeue	2*200	0/80	0/80	0/80	288	726	230	-	-	-	-	-	-
KStack	push;pop	2*50	10/20	18/20	15/20	83	30	62	2	3	3	2323	864	831
TBStack	push;pop	2*100	19/20	20/20	20/20	72	124	60	2	2	2	1596	532	356

Compared to version B, versions D and H perform better on the faults "revealed" long after "triggered" (K-Stack, TBStack).

3) From the results in the localization phase, it can be seen that CGVT delivers the test cases with no more than 4 concurrent operations. Such test cases are quite small-scale and adaptable for a fine-grained analysis.

V. Conclusion

This paper presents a tool, CGVT, for building a small test case that is sufficient to trigger a linearizability fault. CGVT firstly acquires a possibly large-scale concurrent history which has been detected non-linearizable, then locates the concurrent operations causing the linearizability violation, and synthesizes a minimum test case. A D-PCT based technique enhances the CGVT's ability in detecting complicated bugs, and a heuristic rule is introduced for accelerating detection and localization.

As future work, firstly, a tool for fine-grained localization is needed. Given a small test case, this tool ought to enumerate the fine-grained traces and investigate the data races on the shared variables to provide some guidance on bug repair. Secondly, we will try to apply CGVT and this tool to more concurrent programs, and extend their localization abilities from linearizability violation to other concurrency bugs.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China under Grants No.61100069, No.61161130530, and by the National Key Basic Research Program of China under Grant No.2014CB340701.

References

[1] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.

[2] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur, "Framework for testing multi-threaded java programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 3-5, pp. 485–499, 2003.

[3] S. Khoshnood, M. Kusano, and C. Wang, "Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 165–176.

[4] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 446–455.

[5] A. Bouajjani, M. Emmi, C. Enea, and J. Hamza, "Tractable refinement checking for concurrent objects," *Acm Sigplan Notices*, vol. 50, no. 1, pp. 651–662, 2015.

[6] A. Horn and D. Kroening, "Faster linearizability checking via p-compositionality," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2015, pp. 50–65.

[7] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," *Acm Sigplan Notices*, vol. 47, no. 6, pp. 521–530, 2012.

[8] Z. Long and Y. Zhang, "Checking linearizability with fine-grained traces," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM, 2016, pp. 1394–1400.

[9] S. Qadeer, A. Sezgin, and S. Tasiran, "Back and forth: Prophecy variables for static verification of concurrent programs," *Tech. Rep. MSR-TR-2009-142*, 2009.

[10] M. Vechev and E. Yahav, "Deriving linearizable fine-grained concurrent objects," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 125–135, 2008.

[11] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: a complete and automatic linearizability checker," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 330–340.

[12] J. M. Wing and C. Gong, "Testing and verifying concurrent objects," *Journal of Parallel and Distributed Computing*, vol. 17, no. 1-2, pp. 164–182, 1993.

[13] G. Lowe, "Testing for linearizability." PODC, 2015.

[14] M. Emmi, C. Enea, and J. Hamza, "Monitoring refinement via symbolic reasoning," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 260–269.

[15] J. E. Gottschlich, G. A. Pokam, C. L. Pereira, and Y. Wu, "Concurrent predicates: A debugging technique for every parallel programmer," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 331–340.

[16] C.-S. Park and K. Sen, "Concurrent breakpoints," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 331–332.

[17] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 167–178.

[18] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[19] A. Haas, T. Htter, C. M. Kirsch, M. Lippautz, M. Preishuber, and A. Sokolova, "Scal: A benchmarking suite for concurrent data structures," 2015.

[20] E. Ladan-Mozes and N. Shavit, *An Optimistic Approach to Lock-Free FIFO Queues*. Springer Berlin Heidelberg, 2004.

[21] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele Jr, "Dcas is not a silver bullet for nonblocking algorithm design," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004, pp. 216–224.