Interleaving-tree Based Localization of Linearizability Faults on the Fine-grained Level

Yang Chen[1,2], Zhenya Zhang[3⋆], Peng Wu[1,2], and Yu Zhang[1,2]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2] University of Chinese Academy of Sciences
[3] National Institute of Informatics

Abstract. Linearizability is an important correctness criterion for concurrent objects. Existing work mainly focuses on linearizability verification of coarse-grained traces with operation invocations and responses only. However, when linearizability is violated, such coarse-grained traces do not provide sufficient information for reasoning about the underlying concurrent program faults. In this paper, we propose a notion of *critical data race sequence* (*CDRS*), based on our fine-grained trace model, to characterize concurrent program faults that cause violation of linearizability. We then develop a labeled tree model of interleaved program executions and show how to identify *CDRS*es and localize concurrent program faults automatically with a specific node-labeling mechanism. We also implemented a prototype tool, FGVT, for real-world Java concurrent programs. Experiments show that our localization technique is effective, i.e., all the *CDRS*es reported by FGVT indeed reveal the root causes of linearizability faults.

Keywords: Linearizability · Bug localization · Concurrency · Testing

1 Introduction

Localization of concurrency faults has been a hot topic for long time. Multiple trials for the same concurrent programs with the same inputs usually result in different outputs. This nature of concurrent program executions is known as nondeterminism. Due to it, it is non-trivial to decide whether a concurrent program is potential to go wrong. Moreover, even if a concurrent program has been known buggy, it is difficult to reproduce the same fault or address the root cause of the fault.

There have been many techniques addressing the problem of localization of concurrency fault. The very basic way is to exhaust the *thread scheduling* space to replay and analyze the buggy execution. Thread scheduling is usually described as a sequence of thread identifiers that reflects the order of thread executions and context switches. In [5, 6, 23], the thread schedule in one buggy execution is

⋆ The work was partially done when the author was a student at Institute of Software, Chinese Academy of Sciences

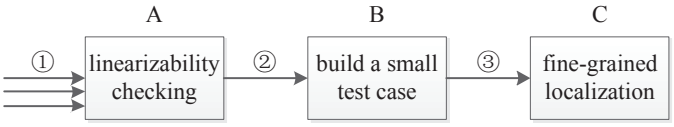
recorded and then used to reproduce the same bugs during the rerun. In many literatures, there is also another notion called *fine-grained trace* that is usually defined as a sequence of memory access instructions and also corresponds to one specific thread schedule. In [12], fine-grained traces and correctness criteria are encoded as logical formulas to diagnose and repair concurrency bugs through model checking. Generally, such fine-grained investigation suffers from a serious problem of state space explosion, and there are many works specifically addressing this issue to accelerate the localization process, such as the heuristic rules based work [2], iterative context bounding [17]. However, the targets of most of the aforementioned works are general concurrency faults, without consideration towards the nature of some specific concurrency fault.

In this paper, we consider the linearizability correctness criterion. Linearizability [10] is a widely-accepted correctness criterion for concurrent data structures or concurrent objects. Intuitively, it means that every operation on a shared object appears to take effect instantaneously at some point, called *linearization point*, between the invocation and response of the operation, and the behavior exposed by the sequence of operations serialized at their linearization points must conform to the sequential specification of the shared object.

Linearizability verification is a hot topic in the recent years' research of concurrency. There are large numbers of works and tools such as [3, 4, 3, 11, 14, 15], focusing on the linearizability verification of the given traces based on a coarse-grained trace model. Usually the traces in such works are described as a partial order set of object methods, and the basic approach for linearizability checking is to enumerate the possible topologically sorted sequential traces that satisfy the correctness criteria of the specific data structures. This approach also suffers from the state space explosion problem in the other level compared to the fine-grained localization work, and thus most of the existing works propose acceleration strategies to address this problem. Furthermore, since the coarse-grained model only investigates composed of method invocations and responses, no information about root causes of linearizability faults is provided by these techniques. Generally, these techniques emphasize acceleration towards the state space explosion problem rather than analysis or localization of linearizability faults.

This paper mainly addresses the issue of the relation between data race and linearizability faults. We propose a notion called *Critical Data Race Sequence* (*CDRS*) based on a fine-grained trace model. Intuitively, a CDRS contains a sequence of data races that decisively result in linearizability faults, and thus the existence of CDRSes implies that the concurrent program is potential to produce non-linearizable traces. Furthermore, in order to identify CDRSes, we model all the possible fine-grained traces of a concurrent execution as an *interleaving tree*, where each node corresponds to a data race and each path from the root to a leaf node corresponds to a fine-grained trace. We label each node with a predefined symbol depending on the linearizability of all the paths passing through the node, and then the existence of CDRSes can be determined by the certain pattern of node sequences in the labeled interleaving tree.

In order to resolve the state space explosion problem, we divide the localization process into a coarse-grained level and a fine-grained level. The coarse-grained level addresses the issue of working out a test case that contains a small number of operations but is sufficient to trigger a linearizability faults [25]. Then, given the small test case as the input for the fine-grained level localization, the number of memory access instructions that should be investigated can be hugely reduced. Together with a linearizability checking technique [15] and coarse-grained localization [25], the overall process is illustrated as in Fig.1, in which C is the main contribution of this work.



① concurrent traces generated by calling the given concurrent data structure
② a concurrent trace that violates linearizability
③ concurrent operations that can trigger linearizability faults

Fig. 1. Labels of nodes

Contributions. The contributions of this paper are as follows:

- We extend the traditional coarse-grained trace model to a fine-grained trace model by adding memory access events. We also extend the notion of linearizability onto fine-grained traces.
- We propose the key notion of *Critical Data Race Sequence* (CDRS) for characterizing the data races that are decisive to linearizability faults.
- We develop a labeled interleaving tree model that contains all the possible fine-grained traces of a concurrent execution. Each node corresponds to a data race and is labeled in a way that can reflect the existence of CDRS through a certain pattern.
- We implement a prototype tool, *FGVT*, for real-world Java concurrent programs. Experiments show that FGVT is effective, in that all the CDRSes reported by FGVT indeed reveal the root causes of linearizability faults.

Related work. Automated linearizability checking algorithms were presented in [4, 24], but suffered from a bottleneck of performance. Based on [24], optimized algorithms were proposed in [15] and [11] through partial order reduction and compositional reasoning, respectively. Model checking was applied in [3, 7] for linearizability checking, with simplified first-order formulas that can help improve efficiency. Fine-grained traces were introduced in [14] to accelerate linearizability checking. All these work lays a firm foundation for the localization of linearizability faults.

Efforts have been devoted to concurrency bug localization, e.g., through an active-testing approach based on bug patterns. The potential bug locations can

then be ranked with the suspicious bug patterns gathered. The characteristics of bug patterns were discussed in [16, 8] in details. Memory access patterns were proposed in [20, 21, 13] for ranking bug locations. A fault comprehension technique was further presented in [19] for bug patterns. Definition-use invariants were presented in [22] for detecting concurrency bugs with pruning and ranking methods. A constraint-based symbolic analysis method was proposed in [12] to diagnose concurrency bugs.

Some other concurrency bug localizations techniques were based on the bug localization techniques for sequential programs. In [9], concurrent predicates were derived from an assertion mechanism to determine whether a data race causes a concurrency bug. Concurrent breakpoints, an adaption of a breakpoint mechanism, were proposed in [18] for concurrent program debugging.

We claim that the novelty of this work is that we focus on the correctness criterion of linearizability, and thus tree search also focuses on such a direction. We apply some existing tree search techniques such as partial-order reduction but we emphasizes our novel approach of localizing linearizability faults.

Organization. The rest of the paper is organized as follows. Section 2 presents an example to illustrate our motivation. Section 3 introduces our fine-grained trace model. Section 4 presents the key notion of CDRS based on our fine-grained trace model. Section 5 shows the labeled interleaving tree model, and the pattern of CDRSes. Section 6 reports the implementation and experiments about our prototype tool FGVT. Section 7 concludes the paper.

2 Motivating Example

In this section, we illustrate the motivation of this work through a buggy concurrent data structure *PairSnapShot* [14].

Fig.2 shows a simplified version of PairSnapShot, where it holds an array `d` of size 2. A `write(i,v)` operation writes `v` to `d[i]`, while a `read` $\rightarrow \langle v_0, v_1 \rangle$ operation reads the values of `d[0]` and `d[1]`, which are v_0 and v_1, respectively.

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

Fig. 2. A concurrent data structure: PairSnapShot

A correctness criterion of PairSnapShot is that `read` should always return the values of the same moment. However, Fig. 3 shows a concurrent execution in which the return values of `read` do not exist at any moment of the execution. In Fig. 3, time moves from left to right; dark lines indicate the time intervals of operations and the short vertical lines at the both ends of a dark line represent the moment when an operation is invoked and returned, respectively.

A label $t : (v_0, v_1)$ indicates that at the moment t, `d[0]` is v_0 and `d[1]` is v_1. The operation `read()` on Thread 2 returns a value $\langle 1, 2\rangle$, which is not consistent with value of any moment.

The reason of this violation can be found out by enumerating the possible executing orders of memory access events which is labeled by `#` in Fig. 2. One possible order that can trigger the violation is illustrated in Fig. 3, in which "`x`" indicate the executing moments of the corresponding memory access events. Actually, this model checking approach is the most common way to locate the root cause of concurrency bugs, and has been studied in many existing literatures. Here, our focus is not on how to find this fine-grained executing order, but to study how the thread execution order, in which data race exists, influences the final result of linearizability.

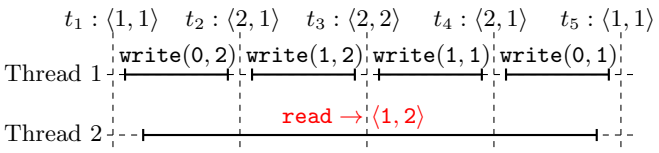


Fig. 3. A buggy trace of PairSnapShot

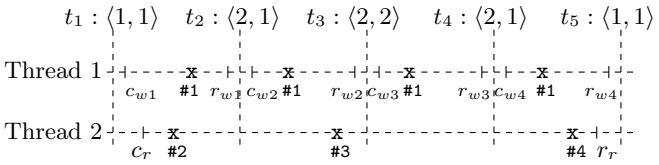


Fig. 4. An executing order of memory access events triggering the violation in Fig. 3

3 Preliminary

In this section, we extend the traditional coarse-grained trace model [3], recalled in Section 3.1, to the fine-grained trace model presented in Section 3.2. Compared to the traditional one, our new model includes memory access instructions such as `read, write` and atomic primitive *compare-and-swap* (`CAS`). This enables us to reason about the causes of linearizability faults on the fine-grained level.

3.1 Coarse-grained trace model

A *trace* S is a finite sequence of events, and each event is in the form $e(Arg)^{\langle o,t\rangle}$, where e is an event symbol ranging over a pre-defined set E, Arg represents arguments, o belongs to a set O of operation identifiers and t belongs to a set T of thread identifiers. In our coarse-grained trace model, the set E contains the following subsets:

- C contains symbols that represent operation invocation events. An invocation event is represented as $c(v_a)^{\langle o,t\rangle}$ ($c \in \mathsf{C}$), where v_a is the argument of the operation;
- R contains symbols that represent operation response events. A response event is represented as $r(v_r)^{\langle o,t\rangle}$ ($r \in \mathsf{R}$), where v_r is the return value of the operation.

We also use C, R to represent the set of corresponding events indiscriminately, and symbol $e \in \mathsf{C} \cup \mathsf{R}$ to represent an event. The order relation between events in a trace S is written as $\prec_S$ (or $\prec$), i.e., $e_1 \prec_S e_2$ if e_1 is ordered before e_2 in S. We denote the operation identifier of an event e as $\mathtt{op}(e)$, and thread identifier as $\mathtt{td}(e)$. An invocation event $c \in \mathsf{C}$ and a response event $r \in \mathsf{R}$ *match* if $\mathtt{op}(c) = \mathtt{op}(r)$, written as $c \diamond r$. A pair of matching events forms an operation with an operation identifier in O, and we represent an operation as $m(v_a) \to v_r$, where m is the operation name. A trace $S = e_1 e_2 \cdots e_n$ is *well-formed* if it satisfies that:

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

A well-formed trace S can also be treated as a partial order set $\langle S, \sqsubset_S \rangle$ of operations on *happen-before* relation $\sqsubset_S$ between operations, where S is called a *coarse-grained trace* (or *coarse-grained history*). The *happen-before* relation $\sqsubset_S$ is defined as that: assuming two operations O_1, O_2 in S are formed by c_1, r_1 and c_2, r_2 respectively, then $O_1 \sqsubset_S O_2$ if and only if $r_1 \prec c_2$.

Example 1. Fig. 3 shows such a well-formed trace: $S = c_{w1}c_r r_{w1} c_{w2} r_{w2} c_{w3} r_{w3} c_{w4} r_r r_{w4}$, where c_{wi} and r_{wi} represents the invocation and response events of the i-th `write` operation respectively, and c_r and r_r represent the invocation and response events of the `read` operation.

It is obvious that $\mathtt{write}(0,2) \prec \mathtt{write}(1,2) \prec \mathtt{write}(1,1) \prec \mathtt{write}(0,1)$, but there is no *happen-before* relation between the `read` operation and any of the `write` operations.

A coarse-grained trace S is *sequential* if $\sqsubset_S$ is a total order. We define that a *specification* of an object is the set of all sequential traces that satisfy the correctness criteria of that object. Note that here correctness criterion is specified by concurrent data structures, such as *first-in-first-out* for *FIFO-Queue*, *first-in-last-out* for *Stack*.

Definition 1 (Linearizability) *A coarse-grained trace S of an object is linearizable if there exists a sequential trace S' in the specification of the object such that:*

1. *$S = S'$, i.e., operations in S and S' are the same;*
2. *$\sqsubset_S \subseteq \sqsubset_{S'}$, i.e., given two operations O_1, O_2 in S, S', if $O_1 \sqsubset_S O_2$, then $O_1 \sqsubset_{S'} O_2$.*

Note that this definition speaks only complete traces, neglecting the existence of pending operations, that is, the operations without response events. Since this paper focuses on analysis of linearizability faults rather than detection, we consider complete traces only.

Example 2. Fig. 5 shows five sequential traces that satisfy requirements 1) and 2) in Definition 1 with respect to the coarse-grained trace shown in Fig 3. However, neither of them satisfies the correctness criteria of *PairSnapShot*, which means that neither of them belongs to the specification set of *PairSnapShot*, so we can say that the coarse-grained trace in Fig. 3 is non-linearizable.

$1 : \texttt{read} \rightarrow \langle 1,2\rangle\ \texttt{write}(0,2)\ \texttt{write}(1,2)\ \texttt{write}(1,1)\ \texttt{write}(0,1)$
$2 : \texttt{write}(0,2)\ \texttt{read} \rightarrow \langle 1,2\rangle\ \texttt{write}(1,2)\ \texttt{write}(1,1)\ \texttt{write}(0,1)$
$3 : \texttt{write}(0,2)\ \texttt{write}(1,2)\ \texttt{read} \rightarrow \langle 1,2\rangle\ \texttt{write}(1,1)\ \texttt{write}(0,1)$
$4 : \texttt{write}(0,2)\ \texttt{write}(1,2)\ \texttt{write}(1,1)\ \texttt{read} \rightarrow \langle 1,2\rangle\ \texttt{write}(0,1)$
$5 : \texttt{write}(0,2)\ \texttt{write}(1,2)\ \texttt{write}(1,1)\ \texttt{write}(0,1)\ \texttt{read} \rightarrow \langle 1,2\rangle$

Fig. 5. Five possible sequential traces

3.2 Fine-grained trace model

In addition to C and R, the symbol set E has other subsets of events in our fine-grained trace model:

- Wr contains symbols that represent the *memory writing* events. An event of memory writing is represented as $wr(addr, v)^{\langle o,t\rangle}$ ($wr \in \mathsf{Wr}$), where $addr$ is the memory location to be modified to a value v;
- Rd contains symbols that represent the *memory reading* events. An event of memory reading is represented as $rd(addr)^{\langle o,t\rangle}$ ($rd \in \mathsf{Rd}$), where $addr$ is the memory location to be read;
- CAS contains symbols that represent the *atomic primitive*, such as *compare-and-swap* (CAS) in modern architecture. A CAS event can be represented as $cas(addr, v_e, v_n)^{\langle o,t\rangle}$ ($cas \in \mathsf{CAS}$), where $addr$ represents a memory location, v_e and v_n are two values. The function of this atomic primitive is that if the value at $addr$ equals to v_e, it will be updated to v_n and return `true`, otherwise it would not do anything but return `false`.

Similarly, $\mathsf{Wr}, \mathsf{Rd}, \mathsf{CAS}$ represent the corresponding sets of events. Let $\mathsf{M} = \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$, events e such that $e \in \mathsf{M}$ are called *memory access events*.

A fine-grained trace S_f is a total order set $\langle S_f, \prec\rangle$ of events over set $E = \mathsf{C} \cup \mathsf{R} \cup \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$. We define a projection $\mathtt{F}_c$ that maps a fine-grained trace S_f to a coarse-grained trace S_c by dropping all memory access events, i.e., $\mathtt{F}_c(S_f) = S_f|_{\{\mathsf{C},\mathsf{R}\}}$. A fine-grained trace S_f is *well-formed* if it satisfies that:

- $\mathtt{F}_c(S_f)$ is well-formed;
- The operation identifier of any memory access event in S_f is consistent with that of one pair of matching invocation/response events in S_f, i.e., $\forall e_m \exists e_o.(\mathtt{op}(e_m) = \mathtt{op}(e_o))$, where e_m is a memory access event and e_o is either of a pair of matching invocation/response events in S_f.
- All memory access events with operation identifier o lie between the consistent matching invocation/response event pair with the operation identifier o, i.e., $\forall e.((\mathtt{op}(e) = o) \rightarrow (c \prec e \prec r))$, where e is a memory access event in S_f, and c, r are matching invocation and response events with thread identifier o.

We claim that all fine-grained traces in this paper are well-formed.

Example 3. Fig.4 presents a well-formed fine-grained trace:
$S_f = c_{w1}c_r\mathtt{\#2\#1}r_{w1}c_{w2}\mathtt{\#1\#3}r_{w2}c_{w3}\mathtt{\#1}r_{w3}c_{w4}\mathtt{\#1\#4}r_r r_{w4}$
and the $\prec$ relation is shown in an obvious manner in the figure. Application of $\mathtt{F}_c$ to this fine-grained trace results in the coarse-grained trace in Fig.3.

Definition 2 *The linearizability of S_f depends on the linearizability of $\mathtt{F}_c(S_f)$, i.e., if $\mathtt{F}_c(S_f)$ is linearizable, then S_f is linearizable.*

We define a predicate $\mathtt{L_n}$ to denote the linearizability of S_f, i.e., if S_f is linearizable, then $\mathtt{L_n}(S_f)$ is true.

4 Critical Data Race Sequence

In this section, we will analyze linearizability faults on the fine-grained level, and propose *critical data race sequence* which is treated as the root causes of linearizability faults.

Definition 3 (Concurrent program) *Each coarse-grained trace S_c results from a concurrent program $\mathbb{P}$. The concurrent program $\mathbb{P}$ is defined as a set of fine-grained traces. Each fine-grained trace $S_f \in \mathbb{P}$ can be mapped to a coarse-grained trace S'_c, which satisfies that:*

- *$S'_c = S_c$, i.e., operations in S'_c and S_c are the same;*
- *$\sqsubset_{S_c} \subseteq \sqsubset_{S'_c}$, i.e., S'_c preserves the happen-before relation in S_c.*

Intuitively, a program $\mathbb{P}$ maintains all fine-grained traces that preserve the *happen-before* relation of S_c. If there is a *happen-before* relation between each pair of operations in S_c, we say that the program $\mathbb{P}$ resulting in S_c is a sequential program. And, if there exists a fine-grained trace that is not linearizable in a program $\mathbb{P}$, we say that the program $\mathbb{P}$ is non-linearizable.

Definition 4 (Linearizability fault) *Let $\mathbb{P}$ be a non-linearizable concurrent program. Each non-linearizable fine-grained trace S_f in $\mathbb{P}$ defines a linearizability fault $\mathcal{F}$.*

We define the prefix relation $\subseteq_{pre}$ between two fine-grained trace S_1 and S_2, that is, $S_1 \subseteq_{pre} S_2$ represents that S_1 is a prefix of S_2. We use $\bullet$ to represent the concatenation of a sequence of events and another sequence of events, that is, if $S_1 = e_1 \cdots e_n$ and $S_2 = e_{n+1} \cdots e_{n+m}$, then $S_1 \bullet S_2 = e_1 \cdots e_n e_{n+1} \cdots e_{n+m}$.

Definition 5 (High-level data race [1]) *Let $\mathbb{P}$ be a concurrent program. A high-level data race (HLDR) D in $\mathbb{P}$ is defined as a triple $\langle Var, SE, CE \rangle$. Here, Var is a set containing one or more shared variables, each corresponding to a memory location. SE is a sequence of events, and the execution of SE leads Var to an initial state. CE is a set containing at least two memory access events, such that:*

- *each event belongs to a distinct thread identifier;*
- *each event accesses some shared variables in Var;*
- *no temporal relation between each pair of the events;*
- *for any permutation S_p of events in CE, there exists a fine-grained trace $S \in \mathbb{P}$ such that $SE \bullet S_p \subseteq_{pre} S$.*

Note that here the decision of *Var* depends on the algorithm of the shared object. The most common situation is that several events simultaneously access the same memory location and do some read or modification. However, there also exist other situations, such as *PairSnapShot* in Section 2, in which case several memory locations should be considered as a whole.

Given a HLDR $\mathsf{D} = \langle Var, SE, CE \rangle$ where $e_1, e_2 \in CE$, we say e_1 *wins* e_2 with respect to a fine-grained trace S_f if $SE \bullet e_1 e_2 \subseteq S_f$.

Given a $\mathbb{P}$, we define a partial order relation $<_{dr}$ between two HLDRs $\mathsf{D}_1 = \langle Var_1, SE_1, CE_1 \rangle$ and $\mathsf{D}_2 = \langle Var_2, SE_2, CE_2 \rangle$, that is, if $SE_1 \subseteq_{pre} SE_2$, then $\mathsf{D}_1 <_{dr} \mathsf{D}_2$.

Theorem 1 *If there is a linearizability fault, then there is a high-level data race.*

Proof. To prove Theorem 1, it suffices to prove the contrapositive proposition that if there is no high-level data race, then there is no linearizability fault. According to Definition 5, the premise, no high-level data race, means that in *CE*:

$$\exists S_p \forall S (SE \bullet S_p \nsubseteq_{pre} S)$$

Here, if $\mathbb{P}$ is a concurrent program, this condition is not satisfied according to Definition 3. Therefore, the $\mathbb{P}$ that satisfies this condition corresponds to a sequential program, and the sequential trace surely has no linearizability fault.

Definition 6 (Critical data race sequence) *Let $\mathbb{P}$ be a concurrent program, $\mathcal{F}$ be a linearizability fault. A Critical Data Race Sequence (CDRS) with respect to $\mathcal{F}$ is a total order set of data races $\{\mathsf{D}_1, \mathsf{D}_2, \cdots, \mathsf{D}_n\}$*

$$\begin{aligned} \{\ & \langle Var_1, SE_1, CE_1 = \{e_{11}, e_{12}, \cdots, e_{1m_1}\} \rangle, \\ & \langle Var_2, SE_2, CE_2 = \{e_{21}, e_{22}, \cdots, e_{2m_2}\} \rangle, \\ & \qquad \vdots \\ & \langle Var_n, SE_n, CE_n = \{e_{n1}, e_{n2}, \cdots, e_{nm_n}\} \rangle\ \} \end{aligned}$$

where the relation $<_{dr}$ *exists as* $\mathsf{D}_1 <_{dr} \mathsf{D}_2 <_{dr} \cdots <_{dr} \mathsf{D}_n$. *A CDRS satisfies that there exist two events* $e_{i1}, e_{i2} \in CE_i$ $(i \in 1, \ldots, n)$ *that* e_{i1}*'s win and* e_{i2}*'s win lead the program to "inverse" consequences. Here the meaning of "inverse" includes that*

- *If all the fine-grained traces* S_{f1} *such that* $SE \bullet e_{i1} \subseteq_{pre}$ *are linearizable, then there exist fine-grained traces* S_{f2} *such that* $SE_i \bullet e_{i2}$ *that are non-linearizable;*
- *If all the fine-grained traces* S_{f1} *such that* $SE \bullet e_{i1} \subseteq_{pre}$ *are non-linearizable, then there exist fine-grained traces* S_{f2} *such that* $SE_i \bullet e_{i2}$ *that are linearizable.*

Intuitively, a *CDRS* contains all the HLDRs which are decisive to the linearizability of the trace. Note that although different *CDRS*es in a program $\mathbb{P}$ may lead to different linearizability faults, what we focus on is just the linearizability of the trace and thus we consider all linearizability faults identical in terms of the aspect to lead the trace non-linearizable.

Example 4. Take a look at the example of a HLDR $\mathsf{D} = \langle Var, SE, CE \rangle$ in *PairSnapShot* in which $CE = \{\texttt{\#1}, \texttt{\#2}\}$. If `#1` wins, then a non-linearizable trace will never occur; but if `#2` like Fig. 4, there exists at least such a fine-grained trace that is non-linearizable. In this sense, D is included in a CDRS with respect to the linearizability fault $\mathcal{F}$ shown in Fig. 4.

5 Identify CDRS on Interleaving Tree

From Section 4, we know that it is the competitions happening in *CDRS*es that trigger the linearizability faults. In order to identify *CDRS*, we propose an approach based on a model called labeled interleaving tree in this section. Firstly, we represent the fine-grained traces in an interleaving tree, and then we label the nodes of the tree with a symbol system. We will show that all *CDRS*es follow a certain pattern and thus we can identify them based on the characteristics of nodes.

5.1 Interleaving tree

Firstly, we define a projection $\mathtt{F_M}$ mapping a fine-grained trace S_f to a trace S_f^M composed of only memory access events in S_f, i.e., $\mathtt{F_M}(S_f) = S_f|_{\mathsf{M}}$. Surely the linearizability of S_f^M is consistent with that of S_f. Besides, we define a state, written as *State*, of an object to be a projection mapping memory locations to their values.

Definition 7 (Interleaving Tree) *An Interleaving Tree of* $\mathbb{P}$ *is a tree, where each node corresponds to a state, and each edge corresponds to a memory access event. A subtree rooted at node* N_d *is represented as* $\mathcal{T}ree(N_d)$. *The set of the leaves of the tree is represented as* N_{lf}.

Algorithm 1 presents how to build an interleaving tree recursively. In line 1, *State* is initiated to the initial state of the object. The set *enS* in line 2 initially contains the events which are minimal w.r.t. $\prec$ over the events with the same thread identifier, and thus the number of elements in *enS* is the same as the number of threads. Line 3-11 is the process of building tree. Firstly, a node is built as line 4 shows. Then, events in *enS* are traversed, each accessing a memory location *addr* as line 5 shows. When an event is accessed, a corresponding edge is built as in line 6. Then the state is updated by substituting the value of *addr* with v_n as in line 7. Here note that if e is an Rd event, *State* will not be modified. And *enS* is updated as line 8 shows, where e will be replaced by its successor w.r.t. $\prec$ over the events with the same thread identifier. Finally, the updated *State* and *enS* are applied as arguments to another invocation of BuildTree as line 9 shows to build a subtree recursively.

Algorithm 1 Building of Interleaving Tree

```
1: State = State_init
2: enS = {e | ∀ϵ ∈ M.(td(ϵ) = td(e) ⟶ e ≺ ϵ)}
3: function BuildTree(State, enS)
4:     New Node(State)
5:     for e(addr, v_n)^⟨o,t⟩ ← enS do
6:         New Edge(e)
7:         State ← State[v_n/addr]
8:         enS ← enS \ {e} ∪ {e'}
9:         BuildTree(State, enS)
10:    end for
11: end function
```

Example 5. According to Algorithm 1, we build the interleaving tree of the concurrent program $\mathbb{P}$ corresponding to Fig.3 and present it in Fig.6. Each node represents a state, and each edge represents a memory access event.

Although due to the limitation of space we have omitted many paths, we can still see that this tree maintains all the fine-grained traces in $\mathbb{P}$, and among these traces the one with bold paths corresponds to the non-linearizability situation in Fig.4.

5.2 Identify CDRS on labeled interleaving tree

After building an interleaving tree, we design a symbol system to label the tree in order for the identification of CDRS.

Since each leaf $l_f \in N_{lf}$ corresponds to a fine-grained trace from root to itself, we directly apply $\mathtt{L_n}$

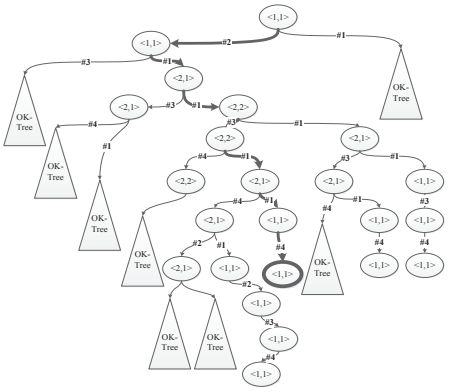


Fig. 6. Interleaving tree of PairSnapShot

to l_f to check the linearizability of the corresponding fine-grained trace. Here, we take a binary interleaving tree built by traces with two threads to illustrate our symbol system.

In our system, a subtree $\mathcal{T}ree(N_d)$ rooted at N_d and holding a leaf set N_{lf} can be grouped into one of the following categories:

- *OK*-tree — all fine-grained traces are linearizable, $\forall l_f\,(l_f \in N_{lf} \rightarrow \mathtt{L_n}(l_f))$
- *ERR*-tree — all fine-grained traces are non-linearizable, $\forall l_f\,(l_f \in N_{lf} \rightarrow \neg\mathtt{L_n}(l_f))$
- *MIX*-tree — both linearizable and non-linearizable fine-grained traces exist, $\exists l_{f1}(l_{f1} \in N_{lf} \wedge \mathtt{L_n}(l_{f1})) \wedge \exists l_{f2}(l_{f2} \in N_{lf} \wedge \neg\mathtt{L_n}(l_{f2}))$

Definition 8 (Node Labeling) *Based on the categories of subtrees, a node N_d can be labeled as one of the following symbols,*

- *W-node — if $\mathcal{T}ree(N_d)$ is an OK-tree.*
- *B-node — if $\mathcal{T}ree(N_d)$ is an ERR-tree.*
- *G-node — if one subtree of N_d is an OK-tree, and the other is an ERR-tree.*
- *GG-node — if two subtrees of N_d are both MIX-trees.*
- *WG-node — if one subtree of N_d is an OK-tree, and the other is a MIX-tree.*
- *BG-node — if one subtree of N_d is an ERR-tree, and the other is a MIX-tree.*

where W represents white, B represents black and G represents grey actually. Fig. 7 illustrates this labeling rule.

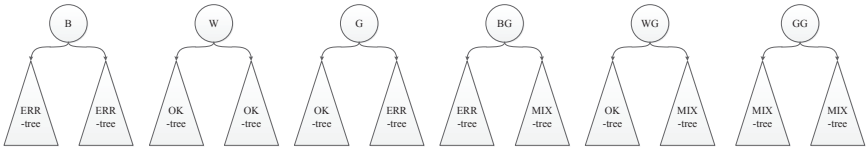


Fig. 7. Labels of nodes

The algorithm of labeling an interleaving tree is presented in Algorithm 2. The function LabelNode recursively labels the nodes of a tree. Firstly, it checks whether the node being labeled has left or right child in line {3,6,8,10}, where $\mathtt{Left}(N_d)$ gets the left child of N_d, and $\mathtt{Right}(N_d)$ gets the right child. $!\mathtt{Left}(N_d)$ means that N_d has no left child, and $!\mathtt{Right}(N_d)$ is in a similar way. So if both $!\mathtt{Left}(N_d)$ and $!\mathtt{Right}(N_d)$ are true, it means N_d is a leaf and thus it is labeled depending on the linearizability of itself as line 3-5 shows. Otherwise, the node is labeled depending its left and right child as line 6-27 shows.

Theorem 2 (Completeness) *Each node of the interleaving tree belongs to one kind of the nodes in Definition 8.*

Algorithm 2 Labeling Interleaving Tree

```
1: Label = {W, B, G, GG, WG, BG}
2: function LabelNode(N_d)
3:     if !Left(N_d)&!Right(N_d) then
4:         if L_n(N_d) then return W
5:         else return B
6:     else if Left(N_d)&!Right(N_d) then
7:         return LabelNode(Left(N_d))
8:     else if !Left(N_d)&Right(N_d) then
9:         return LabelNode(Right(N_d))
10:    else
11:        Label_l = LabelNode(Left(N_d))
12:        Label_r = LabelNode(Right(N_d))
13:        switch ⟨Label_l, Label_r⟩
14:          case ⟨W, W⟩:
15:            return W
16:          case ⟨B, B⟩:
17:            return B
18:          case ⟨B, W⟩|⟨W, B⟩
19:            return G
20:          case ⟨(B|W|G)?G, W⟩|⟨W, (B|W|G)?G⟩:
21:            return WG
22:          case ⟨(B|W|G)?G, B⟩|⟨B, (B|W|G)?G⟩:
23:            return BG
24:          case ⟨(B|W|G)?G, (B|W|G)?G⟩:
25:            return GG
26:        end switch
27:    end if
28: end function
```

Actually, each node N_d of an interleaving tree together with all of its out-edges corresponds to a data race $\mathsf{D} = \langle Var, SE, CE \rangle$. The set Var is a subset of the domain of $State$, where $State$ is represented by the value in a node N_d, SE is a prefix composed of events represented by edges from the root to N_d, and CE contains all events e each corresponding to an out-edge of N_d. Therefore, we can uniquely identify a data race by a node.

Theorem 3 (Identifying CDRS) *A CDRS is equivalent to a subset of nodes in a root-to-leaf path, satisfying a regular expression form*

$$(W_g|B_g)^*(B_g|G)$$

where W_g, B_g, G respectively represent WG-node,BG-node,G-node.

Proof. – Firstly we show that the node sequence following $(W_g|B_g)^*(B_g|G)$ in an interleaving tree is a CDRS. From the definition of W_g-node, B_g-node, and G-node, it is obvious that the HLDRs composed by these 3 kinds of node and their out-edges all belong to the data races described in the Definition 6.
– Then we show that a CDRS appears as $(W_g|B_g)^*(B_g|G)$ in an interleaving tree. According to Definition 6, the two different cases for "inverse" consequences correspond to the W_g-node and B_g-node. Furthermore, since CDRS implies a linearizability fault $\mathcal{F}$, the ending of a CDRS should be that there exists an event whose win can lead all fine-grained trace non-linearizable, and that is just the case of BG-node and G-node, which corresponds to the expression in the theorem.

Example 6. Take a look at Fig. 6. We label the tree according to Definition 8, resulting in a labeled tree presented in Fig. 8. As we can see, the thickened path with a red leaf is non-linearizable, the nodes on which include a CDRS. The CDRS is shown by the sequence of yellow nodes, in the form of $W_g\,W_g\,W_g\,W_g\,W_g\,G$, which is accepted by the regular expression in Theorem 3.

6 Implementation and Evaluation

We have integrated what we presented in Section 5 into a prototype tool called FGVT (Fine-grained VeriTrace), and experiments show that given a *minimum test case* [25], our tool is able to localize the CDRS. In this section, we will give a brief introduction about our tool and experiments, and display the experiment results to show the power of FGVT.

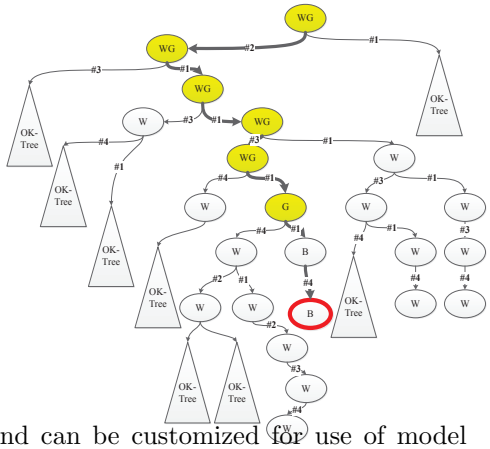


Fig. 8. Labeled interleaving tree.

6.1 Implementation

Our tool FGVT is based on the framework of JavaPathFinder (JPF), which encapsulates a Java virtual machine and can be customized for use of model checking of Java programs. JPF is employed to generate interleaving trees, and it applies a dynamic reduction mechanism to eliminate duplicated program states and simplify the interleaving tree. Then, we label the tree based on Algorithm 2, and report the CDRSes which cause linearizability faults.

6.2 Evaluation

We evaluate our tool by 6 test cases either from prior work or from real applications. The concurrent data structures and how the violations are caused have

Table 1. Evaluation Result

Concur. object	Initial State	Operations	CDRS	Relating data race
LockFreeList	{1}	thd1:`remove(1)` thd2:`remove(1)`	W_gG	`curr.next.get()` `attemptMark()`
OptimisticQueue	{1,2}	thd1:`poll()` thd2:`poll()`	G	`head.getItem()` `casHead()`

PairSnapshot	$\langle 1,1\rangle$	thd1: `write(0, 2)`,
		`write(1, 2)`,
		`write(1, 1)`,
		`write(0, 1)`
		thd2:`read()`
			$W_g\{5\}G$	`d[i]=v`
				`x=d[0],y=d[1]`
				`if(x==d[0])`

Snark	{1}	thd1:`popRight()`
		thd2:`pushRight(2)`,
		`popLeft()`
			W_gW_gG	`rh=RightHat`
				`DCAS(&RightHat,...)`
				`DCAS(&LeftHat,...)`
				`if(rh.R==rh)`

SimpleList	{1}	thd1:`add(3)` thd2:`add(4)`	B_g or B_gG	`pred.next=node` `curr.val<v`
LinkedList	{1,5}	thd1:`remove(1)`, `add(9)` thd2:`size()`	W_gW_gG	`node.next==tail` `synchronized(){...}`

been introduced in last section. In our experiments, all the concurrent objects are executed by two threads, and the operations being tested with initial states of each concurrent object and arguments are listed in Columns 2-3 of Table 1.

The node sequence patterns which are found based on the labeled interleaving tree is listed in Column 4 of Table 1. As we present before, these patterns exactly correspond to the CDRSes of each test case, and here we got some conclusions from this experimental results:

- All the patterns follow the form of regular expression in Theorem 3.
- Most of the test cases end with a G-node, and *SimpleList* shows us a sequence ending with a BG-node.
- We can see the case where not only one CDRS exists.

Column 5 lists the relating source code corresponding to the CDRSes. The source code is acquired from the events participating in the CDRSes, and facilitates the bug repair a lot. For example, we can repair the linearizability faults in *LockFreeList* by transforming `attemptMark` into `compareAndSet`, while there also exist other situations, such as *PairSnapShot*, where we cannot point out exactly the modification of which instructions would lead the object linearizable, since all the data races participate in the CDRSes.

7 Conclusion

This paper proposes the notion of *critical data race sequence* (*CDRS*) that characterizes the root causes of linearizability faults based on a fine-grained trace

model. A CDRS is a set of data races that are decisive to trigger linearizability faults. Therefore, the existence of a CDRS implies that a concurrent execution has potential to be non-linearizable. We also present a labeled interleaving tree model to support automated identification of CDRS. A tool called FGVT is then developed to automatically identify CDRSes and localize the causes of linearizability faults. Experiments have well demonstrated its effectiveness and efficiency.

This work reveals the pattern of the data races that are decisive on the linearizability of a trace. These data races can be mapped to certain parts of the source code. It would be interesting to establish a stronger relationship between CDRSes and the source code for the sake of bug analysis and repair.

References

1. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test., Verif. Reliab. **13**(4), 207–227 (2003)
2. Ben-Asher, Y., Farchi, E., Eytani, Y.: Heuristics for finding concurrent bugs. Proc. IPDPS 2003. pp. 288a–288a (2003)
3. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: SIGPLAN-SIGACT Symposium on 2015, Mumbai, India, January (2015)
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In Proc. PLDI 2010. pp. 330–340 (2010)
5. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. Proceedings of the Sigmetrics Symposium on Parallel & Distributed Tools pp. 48–59 (2000)
6. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded java programs. Concurrency and Computation: Practice and Experience **15**(3-5), 485–499 (2003)
7. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Proc. PLDI 2015. pp. 260–269 (2015)
8. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: Abstracts Proceedings IPDPS 2003. p. 286 (2003)
9. Gottschlich, J.E., Pokam, G., Pereira, C., Wu, Y.: Concurrent predicates: A debugging technique for every parallel programmer. In Proc. PACT 2013. pp. 331–340 (2013)
10. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
11. Horn, A., Kroening, D.: Faster linearizability checking via p-compositionality. In Proc. FORTE 2015. pp. 50–65 (2015)
12. Khoshnood, S., Kusano, M., Wang, C.: Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In Proc. ISSTA 2015. pp. 165–176 (2015)
13. Liu, B., Qi, Z., Wang, B., Ma, R.: Pinso: Precise isolation of concurrency bugs via delta triaging. In Proc. ICSM 2014. pp. 201–210 (2014)
14. Long, Z., Zhang, Y.: Checking linearizability with fine-grained traces. In Proc. SAC 2016. pp. 1394–1400 (2016)
15. Lowe, G.: Testing for linearizability. Concurrency and Computation: Practice and Experience **29**(4) (2017)

16. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proc. ASPLOS 2008. pp. 329–339 (2008)
17. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In Proc. PLDI 2007. pp. 446–455 (2007)
18. Park, C., Sen, K.: Concurrent breakpoints. In Proc. PPOPP 2012. pp. 331–332 (2012)
19. Park, S.: Fault comprehension for concurrent programs. In Proc. ICSE 2013. pp. 1444–1446 (2013)
20. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In Proc. ICSE 2010. pp. 245–254 (2010)
21. Park, S., Vuduc, R.W., Harrold, M.J.: A unified approach for localizing non-deadlock concurrency bugs. In Proc. ICST 2012. pp. 51–60 (2012)
22. Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., Zheng, W.: Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In Proc. OOPSLA 2010. pp. 160–174 (2010)
23. Stoller, S.D.: Testing concurrent java programs using randomized scheduling. Electr. Notes Theor. Comput. Sci. **70**(4), 142–157 (2002)
24. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1-2), 164–182 (1993)
25. Zhang, Z., Wu, P., Zhang, Y.: Localization of linearizability faults on the coarse-grained level. In Proc. SEKE 2017. pp. 272–277 (2017)