Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

2010

Analyzing Concurrency Bugs Using Dual Slicing

Dasarath Weeratunge
Purdue University, dweeratu@cs.purdue.edu

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

William Summer
Purdue University, wsummer@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:

Weeratunge, Dasarath; Zhang, Xiangyu; Summer, William; and Jagannathan, Suresh, "Analyzing Concurrency Bugs Using Dual Slicing" (2010). *Computer Science Technical Reports.* Paper 1734.
http://docs.lib.purdue.edu/cstech/1734

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Analyzing Concurrency Bugs Using Dual Slicing

Dasarath Weeratunge Xiangyu Zhang William N.Sumner Suresh Jagannathan
Department of Computer Science, Purdue University
{dweeratu,xyzhang,wsumner,suresh}@cs.purdue.edu}

Abstract

Because debugging concurrent software failures is so challenging, there has been much interest in developing analyzes to detect concurrency bugs that arise because of data races, atomicity violations, execution omission, etc. However, determining whether reported bugs are in fact real, and understanding how these bugs lead to incorrect behavior, remains a labor-intensive process. This paper proposes a novel dynamic analysis that automatically produces the causal path of a concurrent failure leading from the root cause to the failure. Given two schedules, one inducing the failure and the other not, our technique collects traces of the two executions, and compares them to identify salient differences. The causal relation between the differences is disclosed by leveraging a novel slicing algorithm called *dual slicing* that slices both executions alternatively and iteratively, producing a slice containing trace differences from both runs. Our experiments show that dual slices tend to be very small, often an order of magnitude or more smaller than the corresponding dynamic slices; more importantly, they enable precise analysis of real concurrency bugs for large programs, with reasonable overhead.

1. Introduction

Debugging concurrent software is a challenging exercise because of non-determinism induced by scheduling decisions and thread interactions. There has been much recent interest on developing techniques to identify potential concurrency bugs such as data races [2, 14, 17, 18, 27], atomicity violations [5, 7, 15, 22, 25], or deadlocks [9]. Common to these approaches is a means to check that important properties sufficient to guarantee the absence of a concurrency bug are maintained. For data race detection, this may entail checking that all accesses to the same shared variables are protected by the same set of locks. To ensure the absence of an atomicity violation, an analysis must guarantee that all accesses performed within an atomic section can be serialized with accesses to the same data in other concurrently executing atomic regions. Regardless of how these properties are validated, the onus remains on the programmer to interpret reported violations and to determine whether they truly represent a bug, and if so, how the bug causally results in a run time failure. Understanding causality is particularly important because many concurrency bugs cannot simply be ascribed to the proper lack of synchronization, but rather are a manifestation of complex and often subtle protocol violations [12].

To make concurrency bug detection more useful, we explore the realization of a more general debugging and analysis framework. Given an execution known to manifest a concurrency bug, we perform a postmortem dynamic analysis to *localize* the failure, i.e. our analysis identifies the statement level causal path that leads to the failure. Our approach is not biased towards any specific kind of concurrency error (e.g., data races, atomicity violations, execution omission, etc.)

A significant challenge in defining such an analysis is ensuring that we can deterministically reproduce the failure. Fortunately, recent advances in the field [3, 13, 16, 18] have shown that such a goal can be achieved cost-effectively by systematically exploring a certain set of deterministic schedule permutations. For instance, given a buggy program and failure inducing input, CHESS [13] is able to identify a schedule comprising a bounded sequence of preemptions that induces the failure.

The above mentioned techniques can help construct failure-inducing schedules. They are, however, by themselves insufficient to explain how these schedules cause an observable failure. Consider a concurrency bug we encountered in MySQL described in detail in Section 5.1. The bug is due to an atomicity violation, but most atomicity checkers would find it difficult to detect because the atomic region spans multiple methods, and is thus not easily identified. Although the failure also involves a data race, shared variables are well-protected by locks, and the race only manifests under a specific interleaving. Even though the bug report explains how to produce the failure using thread preemption, the report also notes that it is very difficult to explain the failure because it is hard to associate the schedule perturbation that induces the failure to the final crash. In fact, after the failure-inducing preemption, the program continues to execute 7 million instructions, corresponding to roughly 500K source code line instances. Our analysis facilitates identifying the fault by producing a precise causal execution path consisting of only 23 source line statement instances.

Informally, our technique works as follows. Given two runs, one passing and the other failing as dictated by two deterministic schedules, it produces a sequence of execution points in both runs that are causally related and lead from the program point representing the root cause of the failure to the program point at which the failure is detected. The technique exploits the assumption that the two runs only differ after the first schedule difference by computing their *trace differences*. It then uses a novel slicing algorithm called *dual slicing* that works on both runs alternatively and iteratively to causally connect these trace differences to construct the causal path.

Contributions

- We propose a dynamic analysis to identify the root causes of concurrency failures, given two schedules with one inducing the failure, and the other not.
- We define a new trace comparison technique for concurrent executions. The technique collects traces of a passing and failing run of a program. It then aligns the two traces before they are compared. Trace alignment for large multi-threaded programs is challenging in the presence of loops, recursion, and thread interleavings. These features make simple solutions such as using program counters to determine alignment ineffective. Instead, we leverage *execution indexing* [24] to produce precise trace alignments.

- We devise a new slicing algorithm called *dual slicing* that produces significantly smaller and more accurate slices compared to traditional dynamic slicing techniques. Its improved accuracy stems from its ability to exploit salient dependence information from both passing and failing runs. Consequently, it is able to generate highly precise causal paths for a number of different concurrent program failures including data races, atomicity violations (see Section 4.1), and execution omission errors.
- We evaluate our technique on a set of large realistic open-source multithreaded programs, and show our technique is highly effective in precisely identifying the cause of a concurrency failure.

The remainder of the paper is organized as follows. The next section provides motivation for the problem and an overview of our approach. Section 3 discusses our trace comparison mechanism. Section 4 presents the dual slicing algorithm. Experiments and results are presented in Section 5. Section 6 presents related work and conclusions are given in Section 7.

2. Motivation

To motivate our technique, consider the example shown in Fig. 1. The code snippet is shown in (a), which consists of two threads. The `init()` method executed by thread **T2** is supposed to initialize `x` to 5 before the predicate at line 3 is executed, as shown in the passing run (b). However, as the result of the data race between statements 3 and 8, `x` may be initialized after it is used by the predicate. This results in the false branch being taken, leading to the wrong observable output in the failing run (c).

In order to construct a statement-level causal path of this failure, a naive approach would be to apply dynamic slicing [10]. Given a variable at an execution point, a dynamic slice identifies the statement executions that contribute to the variable's value at this point through data and control dependences. However, dynamic slicing falls short as a useful technique for concurrent failure explanation. Figure (d) represents the slice of the wrong output. Extracting a useful explanation of the failure from this slice is not possible because the slice fails to reveal whether the statements it includes have a benign or visible effect on the fault. For example, although the assignment at statement 2 is included as part of the slice, it is not clear how this assignment contributes to the failure; specifically, the presence of the assignment does not explain *why* the predicate at line 3 remained false even though the intended semantics was that it should be true. In other words, the slice does not convey that it was the absence of the initialization (and thus assignment of x to 5 in statement 8) prior to the conditional test at line 3 that led to the failure. Hence, the root cause of the failure is not present in the slice.

Thus, *the failing run by itself does not contain enough information* to localize the root cause of the failure, and construct a meaningful causal path from the root to the failure point. To incorporate information missing from the failed run, we must examine a passing execution as well to compose a high-quality explanation. Specifically, in a concurrent setting, as long as a failure can be reproduced, we can produce a passing run from the same input using only schedule perturbation; this approach is different from debugging sequential program failures, where passing runs are significantly harder to acquire. Armed with both a passing and failing run, our technique first computes the trace differences between them. In Fig. 1 (b) and (c), two types of differences are identified. Bullets represent value differences, meaning corresponding variables have different values in the two respective runs. Triangles represent flow differences, meaning a statement execution occurs in one run but not the other.

In the next step, we apply a novel slicing technique to construct a failure causal path out of these trace differences. The technique works alternatively and iteratively on both the passing and the failing runs. The basic idea is to compute positive information using the passing run and negative information from the failing run. Positive information tells us that in order to avoid the failure, the program *should have* performed some operation(s). Negative information tells that in order to avoid the bug, the program *should not have* performed some operation(s). A failure explanation is generated by fusing both pieces of information.

In the example, the algorithm starts from the failure point 6 in the failing run, which is a flow difference. Through control dependence, the value difference at line 3 is included. At this point, lines 1 and 2 are not included as they are not trace differences, i.e. they do not contain a faulty value. In order to understand why line 3 contains a faulty value, the algorithm alternates to slice the passing run from line 3. In this process, it identifies that `x` at line 3 in the passing run is data dependent on line 8 while it exhibits a data dependence on line 1 in the failing run. These two lines are included in the slice although they are not trace differences. Since line 3 receives its values from two different otherwise benign definitions in the two respective runs, there must be a data race between 3 and 8. The causal path leading from the root to the failure is now easily derived: "*statement 3 should have received its value from statement 8 which should have executed prior to 3 and after 1. The faulty value read at statement 3 leads to the final faulty output at 6.*" Observe that we use the information from the passing run to prune benign states (line 2) and avoid spurious entries in the path.

Fig. 2 illustrates how dual slicing leverages information gleaned from passing and failing runs *iteratively* to construct a meaningful failure explanation. The code snippet in (a) represents a thread containing predicate `race` that is not properly protected against races. In the passing run (b), `race` at line 3 evaluates to false so that the value of `x` is not updated; this results in the predicate at line 5 evaluating to true, ensuring `y` has the value of 1 at line 7. In the failing run (c), a write to `race` from another thread between the execution of statement 2 and 3 changes the thread's control-flow such that `x` acquires the value 1, causing the branch outcome at statement 5 to yield false. Consequently, `y` has the wrong value of 0 at line 7.

Fig. (e) represents the failure slice of `y` at line 7. It contains only line 7 itself and line 2 due to the data dependence on `y`. Note that line 7 is not control dependent on any other statement. Such a slice clearly does not identify the root cause of the failure. The reason is that the connection between the faulty value at 7 and the faulty predicate outcome at line 5, i.e. *the predicate at 5 should have taken the true branch such that* `y` *would get the correct value 1 at 7*, is not captured.

With dual slicing (Fig. (f)), the algorithm starts with 7 in the failing run. Since 7 is not dependent on any trace differences, the algorithm does not have enough information to proceed further in the failing run, except that it discloses that 7 has a wrong value. The algorithm thus switches to slicing the passing run, and dependence edges $7 \rightarrow 6 \rightarrow 5$ are added as shown in (f). At this point, we cannot go further in the passing run as 5 does not depend on any trace differences, again due to the omission of line 4; the value difference at 5 triggers a switch back to slicing the failing run for the second round. An intuitive interpretation is that by slicing the passing run, we have new information about what is faulty in the failing run. This time, slicing the failing run from 5 adds edges $5 \rightarrow 4 \rightarrow 3$ to the slice. The constructed slice captures the fact that "*Statement 3 should not have had the value of true since that would lead to the execution of statement 4, which in turn produces the faulty value of false at 5. As a consequence, 6 was not executed while it should have been, leading to the final faulty output at 7.*"

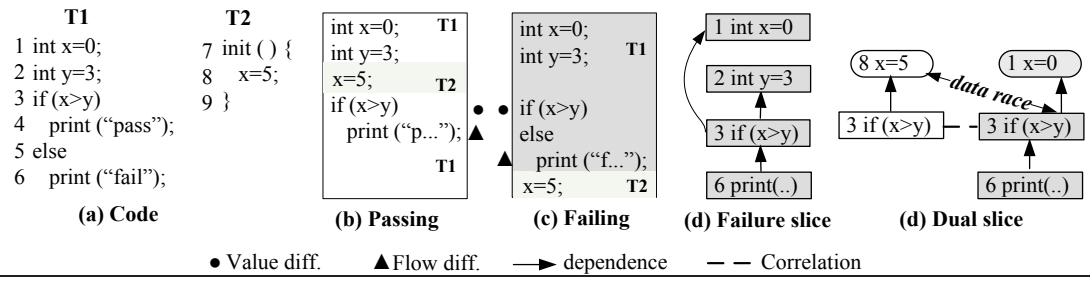


Figure 1. Motivating Example (I) – data race.

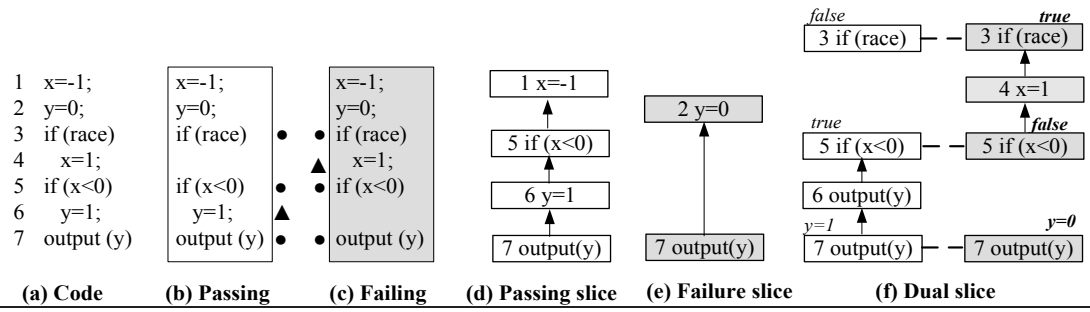


Figure 2. Motivating Example (II) - execution omission.

The mutual symbiosis between the failing and passing run is the key to effectively handling execution omission [29] resulting from concurrency bugs.

3. Trace Comparison

Our technique consists of two phases: trace differencing and dual slicing. Given two runs defined by a correct schedule and a failure-inducing one, the first phase computes their differences. As validated by systems like CHESS [13], in most cases, a failure-inducing schedule can often be derived from a passing one by injecting only a few preemptions. Such preemptions lead to value and/or control differences, some of which are harmful and eventually lead to the failure.

Execution comparison can be carried out on traces. In general, traces maybe either lossy or lossless. Lossless traces [11] record dynamic information for each execution step and thus require space proportional to the execution length. In contrast, lossy traces are acquired by accumulation. For example, a lossy trace captures control flow using program counters, or sometimes more elaborately, as a set of tuples such as *instruction: frequency* and *path: frequency* [1]. The main benefit of lossy traces is that their space requirement can be made linear in static program size. Comparison based on lossy traces is simply comparison of tuples with the same key.

Despite its simplicity and space efficiency, lossy trace comparison is insufficient for our purposes. For example, assume a case in which one run executes a statement s i times, denoted as a frequency pair $s : i$, while the other run executes s $i + 1$ times. Although lossy trace comparison can identify that s is different in the two runs, it fails to identify which instance of s is correlated with the failure. This degree of precision is essential for our purposes. Furthermore, in a concurrent setting, it may happen that a statement s has exactly the same value and execution frequency in both runs and hence is not a trace difference, but it is still strongly correlated to the failure if it is involved in a data race.

Our technique therefore relies on comparing lossless traces. In the context of comparing traces induced by different schedules, the space requirement of lossless tracing is significantly alleviated because we do not need to record any dynamic information *before* the first schedule difference since the two executions are identical before that point. Hence, the main challenge lies in solving the problem of trace alignment. Due to schedule variance, the perturbed execution often makes different function invocations, has different predicate outcomes leading to different control flows, and computes different values for the same variables. If the two traces are not precisely aligned, the computed differences may be due to misalignment, i.e., a trace difference may not be a real difference but instead may be caused by the comparison being carried out at inappropriate points. For example, due to non-determinism, if the same request

```
1  int cnt=2;
2  void main ( ) {
3    Queue reqs;
4    spawn (t_configure( ));
5    while (!reqs.isEmpty( )) {
6      spawn (t_request(reqs.pop( )));
7    }
8  }

20 void t_configure () {
21   if (command=="change count")
22     cnt=readInt();
23   …
24 }

30 void t_ request () {
31   int A[100], B[100];
32   int sum=0, j=0, t_cnt;
33   t_cnt=cnt;
34   while (j<t_cnt) {
35     A[j]=readInt();
36     sum=sum + A[j];
37     B[j]=Integrate(B, j);
38     j++;
39   }
40   if (t_cnt % 2 ==0)
41     sum=sum+A[0];
42   else
43     sum=sum-A[0];
44   output (sum);
45 }
```

data race

Figure 3. A program with a data race. The program creates a configuration thread `t_configure()` and a number of computation threads `t_request()`. A computation thread computes the sum of an input array and the integrals (array `B`) up to each element in the array. Users can set the number of elements to be considered during computation in the configuration thread. The race lies in lines 22 and 33. If, by chance, line 33 is executed before line 22, `t_cnt` receives the old value of `cnt` and leads to unexpected output.

is served by thread t in one run E but by $t+1$ in the other run E' and trace alignment aligns t in E with t in E', the resulting trace differences are meaningless. Therefore, correctly aligning the two traces before they are compared is critical.

To retain the high degree of precision required, execution comparison is therefore performed on lossless traces [11] in which dynamic information is recorded at each execution step.

In this paper, we align two traces of concurrent executions based on their execution structure, leveraging execution indexing [24].

□ **Background : Execution Indexing.**

We use execution indexing [24] to identify the same execution point in both passing and failing executions. The idea is to construct a tree, called the *index tree*, that represents the hierarchical structure of an execution so that executions can be aligned by simply aligning their trees.

Fig. 3 shows a sample program. The program is explained in the caption. Fig. 4 present two executions of the program and their index trees. In the two executions, the user sets `cnt=1` in the configuration thread. In the passing run, the update is successful before the value `cnt` is copied to `t_cnt` in the request thread, whereas in the failing run, the configuration thread is preempted so that `cnt` is not copied to `t_cnt` in time. Let us first focus on the passing execution on the left. The nodes and edges represent the index tree for the trace. The root node denotes the entire execution, which is the body of the main function, represented by node `main`. Informally, this node encapsulates the dynamic scope of the main function. The main body comprises the executions of statements 4 and 5, which are represented by the edges leading from the `main` node to the trace entries representing the statements. Observe statement executions 4 and 5 have their own scopes; their immediate enclosing scope is `main`. Intuitively, the tree captures the fact that execution of the `main` function is composed of two sub-executions, one initiated at statement 4 and the other at statement 5. Note that a statement execution is not part of its own scope. Thus, we introduce an edge between `main` and the trace entry 4 instead of between the internal node "`4 spawn`" and the entry. Similarly, the nesting structure of thread `t_request` is represented by the subtree rooted at "`6 spawn`". That is, thread execution comprises the execution of statement 33, two instances of the while statement 34, the conditional statement 40 and the output statement 44. Furthermore, the first 34 instance and the conditional statement 40 have substructures. The index tree of the failing run is similarly computed and presented on the right. The orientation of the tree is reversed to facilitate easy trace comparison.

Informally, the leaf nodes must be statement executions, i.e. trace entries. An internal node represents a dynamic scope identified by the trace entry that immediately precedes the scope. An edge represents a nest-in relation.

Definition 1. *Given an execution point, represented as a trace entry s, the index of s, denoted as $idx(s)$ is the tree path leading from the root node to s.*

For example, the index of statement instance 22 in the passing run is of "$(main) \rightarrow (4\ spawn) \rightarrow (21\ if) \rightarrow (22)$". The index precisely represents the nesting structure of 22. Traces are aligned by aligning their index trees. More particularly, point x in E aligns with y in E' iff $idx(x) \equiv idx(y)$. Since the index of statement instance 22 in the failing run is the same as that in the passing run, the two 22s align. □

As mentioned earlier, the challenge of trace comparison in the presence of schedule perturbations is that order of statement executions may change, leading to different values produced for the same execution point in the two runs. Intuitively, trace alignment provides a canonical order of statement executions so that comparison can be performed between corresponding (i.e., aligned) execution points, even though they may occur at completely different points in the overall execution order. For instance, statement 22 in the two runs are aligned even though they occur at different places in the traces. Then, the comparison of the aligned 22s determines that statement instance 22 does not have a faulty state in the failing run. In contrast, the second instances of the **while** statement 34 in the two runs are aligned according to the tree alignment. The loop predicate takes a false value in the passing run and a true value in the failing run. Note that the canonicalization induced by the execution index tree does not mean we discard schedule differences. Instead, schedule differences will be faithfully reflected as state differences at aligned points. For instance, it might appear that aligning the instances of 22 incorrectly masks a schedule difference. In fact, the effect of the different schedules is captured by the different values of the aligned instances of statement 33, one of which acquires its value from 22 (in the passing run) and the other (in the failing run) not.

Next, we define trace differences based on execution indexing. We assume the value of each execution instance is also recorded as part of the trace entry. We use $val(s)$ to represent the value of an execution point s. The value of a statement is the value stored in the destination variable. The value of a predicate is its boolean outcome. The value of a method invocation is the return value.

Definition 2. *Given an execution E and a reference execution E', an execution point $s \in E$ is a trace difference if one of the following three conditions is satisfied:*

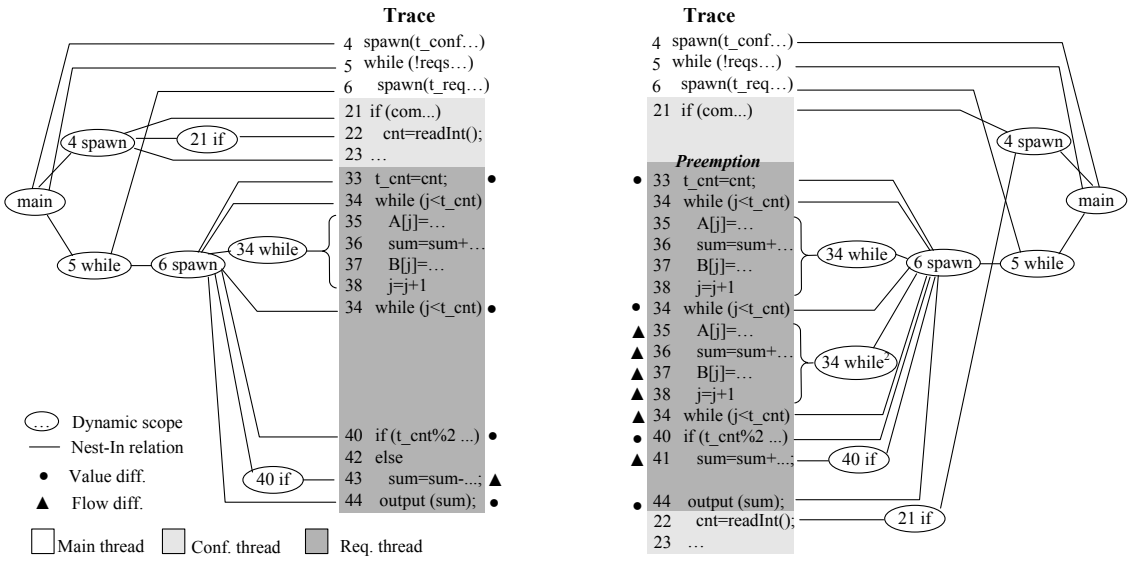
(1) $idx(s)$ is not a valid index in E';
(2) There is an execution point s' in E' such that $idx(s) = idx(s')$, but $val(s) \neq val(s')$.
(3) There is an execution point s' in E' such that $idx(s) = idx(s')$ but s and s' have at least one use which is data dependent on two definitions d and d' where $idx(d) \neq idx(d')$.

If condition (1) is satisfied, s is called a flow difference. If (2) is satisfied, s is called a value difference. If (3) is satisfied it is called a def-use difference. Observe that conditions (2) and (3) may be satisfied simultaneously.

According to the definition, an execution point s is a *flow difference* if it is not aligned with any point in the reference execution. If it does have an alignment but its alignment has a different value, it is a *value difference*. Thus, if s is a value difference, it implies s has an alignment. Finally, if an execution point has an alignment, but some of the subterms of the aligned statement are themselves not aligned, the point is a *def-use difference*.

In Fig. 4, bullets represent value differences and triangles represent flow differences. Statement 33 in the passing run is a value difference because although `t_cnt` has value 1 at this point, it has value 2 at its alignment in the failing run. For the same reason, statement 33 in the failing run is also a value difference. Also note that 33 in the passing run is data dependent on statement 22 whereas it is data dependent on 1 in the failing run (see Fig. 3). This is a def-use difference. Statement 43 in the passing run is a flow difference as it is not aligned with any statement in the failing run. Similarly, statement 41 in the failing run is a flow difference. The second instances of statements 35, 36, 37 and 38 in the failing run are also flow differences.

Observe that our trace differences are defined over statement *instances*, meaning we can identify the specific instance of a statement as a trace difference even though the statement might be executed multiple times in an execution. We discuss how to make use of these trace differences in the next section.



(a) Passing run

(b) Failing run

Figure 4. In these two runs, one configuration thread and one request thread are spawned. In the failing run, the user sets parameter `cnt=1` in the configuration thread, but the thread is preempted by the request thread before the user's change is updated to the variable. As a result, `t_cnt` takes the stale `cnt` value in the request thread.

4. Dual Slicing

Trace differences alone cannot localize the root cause of concurrency failures. To clearly understand a failure, it is necessary to observe a minimum sequence of statement executions that are causally connected, leading from the root cause to the failure. Trace differences often contain excessive redundant information not related to the failure. For example, the second instance of statement 37 in Fig. 4 that assigns to `B[j]` in the failing run is a trace difference. But, it has nothing to do with the observed wrong output at statement instance 44. In our experiments, trace differences for realistic concurrent program executions often subsume 100K or more statements, even though the portion relevant to the failure can be localized to a few tens of statements.

We propose to combine dynamic slicing with trace differencing to identify the root cause of a concurrency failure, and enable construction of the salient execution path from this root to the failure point. Dynamic slicing [10] is a technique that discloses dependences between execution instances and is often used in debugging. A data dependence exists between two statement instances i and j if i writes a value to a variable and that value is used at j, e.g. the first statement instance 34 is data dependent on 33 in the passing run in Fig. 4. A statement instance j is control dependent on a predicate instance p if p directly decides the execution of j. For instance, in the passing run, 43 is control dependent on 40. Given an execution point, its dynamic slice is the transitive closure of the value at that point along dependence edges. Slicing overcomes the aforementioned limitations of trace differencing. More specifically, trace differences can be connected through dependence edges, which essentially represent causality. Redundant information can be pruned by slicing if the failure is not (transitively) dependent on the information. On the other hand, trace differencing substantially improves the effectiveness of dynamic slicing. For example, the slice of statement instance 44 in the failing run includes the first instance of statement 36 because 44 is data dependent on 41, which in turn is data dependent on the second instance of 36 and then the first instance of 36. Similarly, the first instances of 34, 35 and 38 are also in the slice. Using trace differencing, we can easily identify these statement instances as having benign effect, and do not need to include them in the output of the analysis.

Slicing determines the parts of a program "relevant" to some slicing criterion. In traditional slicing, relevance is defined as any statement possibly affecting the values computed by the criterion.. Like thin slicing [19], dual slicing differs from classical slicing primarily in its more selective notion of relevance. Given a trace difference, its dual slice consists of other relevant trace differences. A trace difference d_1 is relevant to another trace difference d_2 if there exist a chain of control and data dependencies from d_1 to d_2 comprising only of trace differences. Hence, unlike a traditional slice, a dual slice does not provide an executable program. For instance, the value difference of a traditional slice necessarily includes the predicate on which it is control dependent. However, if the predicate is aligned in both passing and failing runs, it itself is not a trace difference, and does not need to be included in the dual slice.

A value difference could be the result of the particular statement being data dependent on other value differences or its uses being part of one or more def-use differences. A flow difference could be the result of a particular statement being control dependent on another flow difference or a value difference. However, if we assume the two traces were aligned at the beginning, all flow differences must eventually be control dependent on a value difference. Lastly, a def-use difference could be the result of either a flow difference or a schedule perturbation. If the def-use difference is the result of a flow difference, one or more of its defining statements must be flow differences. However, if it is caused by a schedule perturbation, it is not data or control dependent on any other trace difference.

The algorithm is described in Algorithm 1. For brevity, it assumes data and control dependences are already available. The symbols used are defined in Table 1. The algorithm produces the dual slice of the failure point, represented by two node sets $N_f(N_p)$ and two dependence edge sets $E_f(E_p)$. The subscript represents the failing (f) or passing (p) execution. A node is an execution point identified by its index, e.g. `d` and `t`. A node representing a value difference is in both N_f and N_p.

Input:

- T_f, T_p: the traces of the failing run and the passing run;
- $DTYPE_{f/p}$(s): decides if s is a value difference (VAL_D), a flow difference ($FLOW_D$); the subscripts denote the run.
- $DEP_{f/p}$(s): the set of execution instances that s depends on, including data and control dependences.

Output: $N_{f/p}$, $E_{f/p}$: slice node sets and edge sets.

Note:
isVisitedInPass(s) and *isVisitedInFail(s)* decide if s has been traversed in T_p and T_f.
f_wl and p_wl are worklists for the failing and the passing runs.
`t` and `d` are execution points identified by their index.

Table 1. Symbols used in Algorithm 1.

In the algorithm, the slice node set in the failing run N_f and the failing run worklist f_wl are initialized with the failure point at line 1. The **while** loop in lines 3-27 describes the main dual slicing process. It alternatively and iteratively slices the failing run and then the passing run. Lines 4-23 correspond to slicing the failing run. At line 6, if a value difference is encountered and it has not been encountered in the construction of the *passing* run slice, it is added to the passing run worklist p_wl at line 7, and the passing slice node set N_p at line 8. The **for**-loop in lines 10-22 examines t's dependences. In lines 11-15, the algorithm adds a dependence d to the slice and the work list, if d is a value difference or a flow difference. Otherwise, the algorithm handles *def-use* differences by adding the dependence d to the slice if the dependence d' corresponding to d in the passing run does not align with d (lines 16-21). Observe that though d is added to the slice it is not placed in the work list. Slicing the passing run is symmetric to slicing the failing run and elided for brevity.

```
1  N_f ← f_wl ← {the failure point in T_f} ;
2  N_p ← p_wl ← φ ;
3  while f_wl ≠ φ and p_wl ≠ φ do
4      while f_wl ≠ φ do
5          t ← f_wl.pop() ;
6          if DTYPE_f(t) ≡ VAL_D and !isVisitedInPass(t) then
7              p_wl ← p_wl ∪ {t};
8              N_p ← N_p ∪ {t} ;
9          end
10         foreach d ∈ DEP_f(t) do
11             if DTYPE_f(d) = VAL_D or DTYPE_f(d) = FLOW_D then
12                 N_f ← N_f ∪ {d} ;
13                 E_f ← E_f ∪ {t → d} ;
14                 if !isVisitedInFail(d) then f_wl ← f_wl ∪ {d}
15             else
16                 d' ← The dependence in DEP_p(t) that corresponds to d ;
17                 if d ≠ d' then
18                     N_f ← N_f ∪ {d} ;
19                     E_f ← E_f ∪ {t → d} ;
20                 end
21             end
22         end
23     end
24     while p_wl ≠ φ do
25         Slicing T_p, adding nodes into f_wl. It is symmetric to failure run slicing.
26     end
27 end
```

Algorithm 1: Dual Slicing

Example. The dual slice of the example in Fig. 4 is presented in Fig. 5. During the analysis, the algorithm first adds the index of 44_1, denoting the first instance of statement 44 in the failing trace, to f_wl. The index is then popped from the worklist at line 5 of the algorithm. Since it is a value difference, it is added to p_wl and N_p in lines 7 and 8. Next, the algorithm adds the dependence of 44_1, here 41_1 to the dual slice and f_wl. Since 41_1 is not a value difference, the algorithm simply adds its control dependence 40_1 and the data dependence 36_2 to N_f and f_wl. The index of 40_1 is added to p_wl as well since it is a value difference. The failure slicing loop terminates when 33_1 is reached because 33_1 is not dependent on any trace differences. At this point, all the shaded nodes and their edges as shown in Fig. 5 have been added to the dual slice, and the passing run worklist contains the indices 44_1, 40_1, 34_2, and 33_1. The algorithm switches to slicing the passing run with these criteria. After the passing slicing loop terminates, all the executions represented by plain nodes and their edges are added. This time, no new value differences are added to f_wl and the main computation loop terminates. The rectangular nodes represent either value differences or flow differences. The rounded nodes are non trace differences added to the slice since they define values used by def-use differences. For example, 22_1 and 1_1 are added to the slice since the values they define are used by the def-use difference at 33_1.

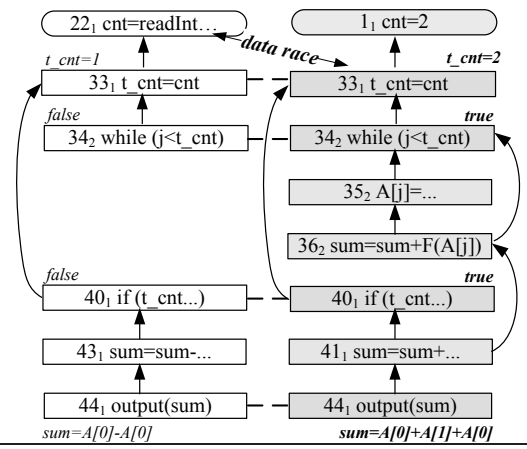


Figure 5. The dual slice of the failure in Fig. 4. A symbol s_i represents the ith instance of statement s in the trace.

4.1 Identifying the Root Cause

Dual slicing is essential to fusing both positive and negative information to better understand failures. Observe that in Fig. 5, the faulty slice indicates *how* a faulty value is been generated at 44_1, but the faulty slice alone is not sufficient to understand *why* this process is faulty. For example, by studying the chain $44_1 \rightarrow 41_1 \rightarrow 40_1$ in the failing run part of the dual slice we can deduce that "40_1 *having the true branch outcome leads to* `sum` *being updated at* 41_1, *and hence induces an observable wrong output*". By analyzing the accompanying positive chain $44_1 \rightarrow 43_1 \rightarrow 40_1$ in the passing run part of the slice, it can be concluded that "40_1 *should have had the false branch outcome, which would have led to the subtraction of* `sum` *at* 43_1, *leading to the correct output* `A[0]-A[0]=0`". Observe that information gathered from the two runs are complementary.

The dual slicing algorithm terminates if it cannot make progress in either run. In the presence of crashes, some flow differences in the passing run may be caused by the failing run being prematurely interrupted, not by different branch outcomes. For example, if the two runs take the same branch but the failing run crashes before

it finishes executing the branch by reaching the immediate post-dominator, unexecuted instructions become flow differences in the passing run. We define them as non-trace-differences (because they would have been present in the trace without the crash, as dictated by the fact that both runs are executing the same branch).

Consider the example in Fig.5. The last trace difference added to the slice is at 33_1 in both runs, which is a def-use difference. The non trace differences 22_1 and 1_1 are added to the slice since they contribute to the def-use difference at 33_1. Specifically, the use of `cnt` at 33_1 has two distinct definitions in the two runs. In the passing run, the definition of `cnt` at 22_1 is used. Instead, in the failing run the definition at 1_1 is used.

Dual slices are particularly useful in debugging concurrency bugs because a correct concurrent execution can be usually obtained easily. This is because concurrency bugs are often the result of def-use differences that arise because of schedule perturbations that expose data races or atomicity violations.

Property 1. *The* root cause *of a concurrency bug is a def-use difference in which there exists a chain of data and control dependencies between the difference and the failure such that each trace difference in the chain is the result of the dependence on its predecessor and if a statement on the chain is control dependent on a trace difference this control dependence is also part of the chain.*

Theorem 1. *The dual slice extracted from a failure point includes the root cause of the failure.*

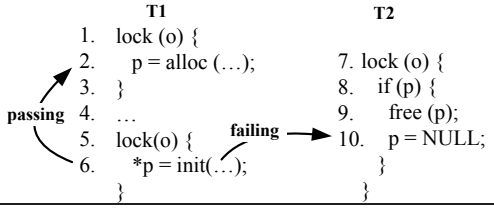


Figure 6. Atomicity violation.

While we have considered how dual slicing can be used to find the root cause of concurrency bugs that arise due to data races and execution omission, it can also be used to identify atomicity violations as well (see Fig. 6). In the example, note that all operations on p are protected by locks and there is no data race. Statements 2 and 6 should be atomic. Otherwise, the remote access 10 may interfere and lead to a null pointer dereference. Assume the dual slice is computed on 6. The dual slice terminates on 6 itself which is a value difference. It is data dependent on 2 in the passing run and on 10 in the failing run. Note that neither 2 nor 10 is a trace difference. 6 and its immediate dependences clearly indicate that 2 and 6 should be atomic and the interference from 10 causes the failure. Deadlock failures are similar.

4.2 Removing Redundancy in Dual Slices

In this section, we explore a more restrictive form of dual slicing. In the previous section, we defined the dual slice as those trace differences that are relevant to the failure. Recall that a value difference cannot be control dependent on a trace difference and it can only be data dependent on other trace differences. Hence, the data dependence directly contributes to the value difference. However, flow differences can be both control and data dependent on other trace differences. Thus far, we have considered both these relationship as being relevant. Assume that the data dependence and the control dependence are part of independent dependence chains. Note that without the control dependence, the dependence chain with the data dependence has no effect on the failure. Hence it cannot lead to the root cause. Thus, we can safely define data dependence chains that fall through flow differences as being irrelevant to a failure.

Consider the dual slice in Fig. 5. In the failing half of the slice, 36_2 is a flow difference. Observe that in order to understand the failure, it is important to know that the execution of 36_2 is decided by the faulty branch outcome at 34_2. In contrast, how the value in 36_2 is computed is not important. Hence, we should consider the control dependence of 36_2 but not its data dependences. As a result, 35_2 can be excluded from the slice. One can also interpret this intuition as follows: since both 35_2 and 36_2 are not (transitively) data dependent on any value differences (i.e., faulty values), the computations of their values are not faulty. What is faulty is the fact that they get executed. To preserve such information, the control dependence from 36_2 to 34_2 is sufficient and hence the dependence from 35_2 to 34_2 is redundant, which justifies removing 35_2 from the slice.

```
 9  /* the same as Algorithm 1. lines 1-9*/;
10  foreach d ∈ DEP_f(t) do
11      if [DTYPE_f(d) = VAL_D ∨ DTYPE_f(d) = FLOW_D]^[1] and
        ((  [DTYPE_f(t) ≡ VAL_D ∧ t --dd--> d]^[2]  ) or
        (   [DTYPE_f(t) ≡ FLOW_D ∧ t --cd--> d]^[3]  ) or
        (   [DTYPE_f(t) ≡ FLOW_D ∧ t --dd--> d ∧
             VAL({d}) − VAL(CD_f(t)) ≠ φ]^[4]       ))
        then
12          N_f ← N_f ∪ {d} ;
13          E_f ← E_f ∪ {t → d} ;
14          if !isVisitedInFail(d) then f_wl ← f_wl ∪ {d}
15      end
16  end
17  /* the same as Algorithm 1. lines 17-21*/
```

Algorithm 2: Optimized Dual Slicing. Edge $\xrightarrow{cd}$ denotes control dependence and $\xrightarrow{dd}$ denotes data dependence. Method *VAL*(s) returns the set of value differences reachable from s. Method *CD*(s) returns the control dependence of s.

Based on the above observation, we propose an optimized dual slicing algorithm, shown in Algorithm 2. The difference from Algorithm 1 lies in the loop (lines 10-16) that adds a trace difference t's dependences into the slice and the worklist. More particularly, the conditions at line 11 controlling the traversal are different. Condition [1] makes sure the dependence d is a trace difference, otherwise it is not traversed. It further controls traversal based on trace difference type and dependence type. Condition [2] specifies that d will be added and traversed if t is a value difference and t is data dependent on d. Condition [3] specifies that if t is a flow difference, its control dependence is added and traversed (data dependences are usually not interesting). Condition [4] specifies an exception: a flow difference's data dependence d may be traversed if d can transitively reach a value difference that can not be reached through the control dependence of t. This means there is a faulty value contributing to t only through d, thus requiring further examination of this dependence. Though condition [4] is not necessary for capturing the root cause, the value differences captured by it help understand the different ways in which the root cause identified by traversing the control dependence in [3] may be affecting the failure. The same optimization is conducted in the passing part of Algorithm 1 except that condition [4] is not considered because

Table 2. Bugs and Tracing.

bug	ID	type	$T_{tracing}$/ T_{slice}	threads	trace size (P/F)(MB)
apache-1	21285	atom	320s/89s	2	1240/1227
apache-2	44402	atom	139s/2742ms	5	326/289
apache-3	45605	race	105s/948ms	2	176/59
apache-4	25520	atom	95s/636ms	2	86/86
mozilla-1	133773	race	74s/1476ms	2	497/477
mozilla-2	342577	race	25s/1353ms	2	290/290
mysql-1	12212	race	218s/623ms	2	110/61
mysql-2	12228	atom	225s/1640ms	2	342/328
mysql-3	12845	atom	133s/589ms	2	83/70
mysql-4	12848	atom	269s/416ms	2	52/62
mysql-5	42419	race	614s/89s	2	567/1297
mysql-6	21587	atom	280s/1637ms	2	72/60
mysql-7	17404	atom	166s/1764ms	3	266/229
pbzip2*	-	race	38s/494ms	2	392/389

race= data race, atom=atomicity violation, *the bug appeared in version 0.9.4

Table 3. Trace Comparison.

bug	t(s)	align % (P,F)	val. diff.	flow diff. (P,F)
a-1	104	(99.7, 100), (99.6, 100)	282, 352	(11k, 151), (12k, 93)
a-2	12	(96.6, 99.7), (99.8, 99.5), (4, 99), (99.8, 96), (96, 99.5)	3k, 40k, 1k, 40k, 3k	(4k, 359), (320, 765), (135k, 55), (221, 5k), (3k, 342)
a-3	1	(96.8, 1.0), (87.2, 66.5)	12, 217	(517k, 42k), (1k, 3k)
a-4	1	(97.5, 97.1), (98.8, 98.8)	690, 586	(3k, 4k), (1k, 1k)
mzl-1	5	(27, 4), (81, 96)	6k, 115	(124k, 1264k), (2k, 329)
mzl-2	2	(76.9, 88.1), (84.5, 69.5)	34, 27	(250, 113), (139, 332)
m-1	6	(48, 93), (58, 95)	385, 377	(164k, 10k), (34k, 2k)
m-2	4	(93, 99), (100, 100)	437, 464	(82k, 9k), (8, 8)
m-3	1	(48, 81), (94, 91)	923, 811	(146k, 31k), (2k, 4k)
m-4	3	(89, 83), (59, 91)	18, 263	(6k, 10k), (36k, 5k)
m-5	50	(83, 75), (60, 86)	1557, 4183	(110k, 184k), (587k, 147k)
m-6	1	(54, 70), (71, 84)	241, 741	(84k, 42k), (19k, 9k)
m-7	5	(67, 95), (99.9, 99.8), (99.6, 99.3)	529, 337, 1307	(149k, 17k), (48, 180), (934, 1594)
pbz	1	(92, 99), (87, 84)	8, 7	(122, 19), (39, 50)

in the passing run, all values are correct, thus making data dependences of a flow difference in the passing run uninteresting.

Property 2. *A pruned dual slice (as defined by Algorithm 2) includes the root cause.*

Example. In the failing part of the slice in Fig. 5, we do not traverse along $36_2 \xrightarrow{dd} 35_2$ as mentioned earlier. In contrast, we traverse along $41_1 \xrightarrow{dd} 36_2$ because $VAL(\{36_2\}) - VAL(\{40_1\}) = \{33_1, 34_2\} - \{33_1, 40_1\} = \{34_2\}$, meaning the value of 41_1 is affected by a faulty value 34_2 exclusively through the dependence $41_1 \xrightarrow{dd} 36_2$, hence requiring the edge to be traversed.

Note that such selective traversal is not applicable for traditional dynamic slicing as it leverages information from trace differencing.

5. Experiments and Results

Our system is implemented using gcc and valgrind. Indexing is implemented in gcc in order to support large multi-threaded programs. The modified gcc compiler instruments a given program to construct and maintain the index tree. Tracing is implemented in valgrind. Trace differencing and dual slicing are implemented in C.

We have collected a pool of 14 real bugs of different types from various sources as shown in Table 2. These bugs are from MySQL (668K LOC), Apache (253K LOC), Pbzip2 (2.9K LOC), and Mozilla-extracts (61K and 3K LOC); their bug report ids are presented in column `ID`. Note that the bugs were collected using full program version for all the benchmarks other than Mozilla, which was reproduced on isolated components[1] Many of the bugs shown here have not been previously studied in any detail in the literature. In this paper, we used the inputs from the reports and reproduced the bugs according to the preemptions mentioned in the reports. Passing runs are acquired by suppressing the preemptions that lead to the bug. In the future, we envision our technique can be integrated with failure inducing schedule generators [3, 13, 16].

Table 4. Slicing

bug	dyn. slice		dual slice			breakdown	fun
	full	bfs	full	opt	bfs	(P/F/vdiff)	
a-1	78k	74	33	9	9	6, 8, 5	5
a-2	20k	695	443	116	106	103, 104, 91	19
a-3	893	113	968	113	41	86, 53, 26	11
a-4	15k	(201)**	99	21	21	20, 21, 20	2
mzl-1	119	3	7	3	2	2, 3, 2	2
mzl-2	346	183	1k	48	34	32, 34, 18	5
m-1	14k	4	7	3	3	2, 2, 1	3
m-2	18k	619	105	43	32	29, 29, 15	5
m-3	5k	101	66	23	19	12, 17, 6	11
m-4	10k	70	75	19	19	17, 13, 11	3
m-5	165k	*	14k	549	121	233, 376, 60	19
m-6	15k	1276	5k	72	60	34, 56, 18	13
m-7	37k	99	131	46	28	28, 42, 24	6
pbz	13	4	7	3	3	2, 2, 1	3

* execution omission leads to root cause vicinity not being reachable.

** another execution omission case: the root cause vicinity can not be reached by slicing trace differences in the failing run, i.e., not reachable through faulty dependences. Interestingly, it can be reached in the dynamic slice through a dependence path that does not contribute to the failure.

Table 2 also shows the cost of our technique including tracing and dual slicing time. The machine used for measurements is an Intel core 2 duo 2.2GHz with 4GB RAM. The number of threads involved in the slice is presented in column `threads`. The aggregate trace size for all threads in the slice is reported in the last column. The required space for each traced instruction is roughly 20 bytes without compression. As mentioned earlier, we start tracing only when the two executions start to diverge, thus avoiding the need to collect the full execution traces. Space costs are less than 1.3 GB across all runs. Tracing and slicing times are all within 11 mins.

Table 3 quantifies the effect of trace differencing. The second column shows the time to collect differences. Each pair of entries in the third column represents the alignment percentage of a thread between the passing (P) and the failing (F) runs. For example, for bug `mysql-5`, 83% of the statements executed by thread one in the passing run align with 75% of the statements executed by the thread in the failing run. The fourth column shows the number of value differences. The last column shows flow differences for each thread. The number of differences are in source code line instance units. Note that except for a few cases, such as `apache-4`, the control flow of the two executions are quite different, reflected by the alignment ratio and the number of flow differences. Observe also that there are many more flow differences than value differences, implying the two runs mainly differ in their control flow. For example, the pair (11k, 151) in the first row (bug a-1) indicates that thread one executes 11k lines exclusively in the failing run and 157 exclusively in the passing one. Thread one of `mzl` shows the other extreme. Both benchmarks reflect widely differing behavior in the passing and failing runs, making it difficult for tools that do not consider both together to localize the root cause of the fault.

Table 4 shows the results of dual slicing and compares it with a traditional dynamic slicing strategy. The second and third columns show the sizes of dynamic slices. Significantly, information gleaned from the dynamic slices extracted from the failing run was not sufficient to identify the root cause of the failure for any of the

[1] Unfortunately, reproducing the bugs on the full Mozilla version requires supporting multiple languages. Our methodology is consistent with other efforts [26].

benchmarks. To reason about why a value is wrong in the failing run, without benefit of examining the passing run, a developer would need to inspect `bfs` number of statements before entering the vicinity of the root cause. We say the vicinity of the root cause is reached if *only one* of the memory accesses involved in a data race, atomicity violation, or deadlock is reached. We assume the traversal proceeds by a breadth-first search from the failure, a common practice in evaluating effectiveness of slicing [19]. The columns under `dual slice` show the data for dual slices, including the `full` dual slice, computed by Algorithm 1, the optimized dual slice (column `opt`), computed by Algorithm 2, and the statements needed to be inspected in the optimized slices (column `bfs`). The breakdown (P/F/vdiff) column shows how many statements in the optimized dual slices belong to the passing run, the failing run, and are value differences. Column `fun` shows how many source code functions span a bfs exploration.

Our results reveal that dynamic slices are substantial in large programs. Note that these executions are not loop intensive, and thus the slices mainly contain executions of unique statements. Even getting to the vicinity of the root cause is non-trivial, e.g., 1276 statements need to be inspected in `mysql-6`. Not surprisingly, dual slices are much smaller than dynamic slices on almost all benchmarks, in many cases, requiring an order of magnitude fewer statements to be inspected. Note also that the optimized algorithm is very effective in pruning redundancy in large dual slices. For example, `mysql-5` is an omission case in which a substantial piece of code is executed in only the passing run but not the failing one; slicing the failing run does not lead to the root cause. The original dual slicing algorithm includes a large portion of the omitted execution into the slice because they are transitively involved in computing the criterion value. However, such information is redundant in identifying the failure point. Our optimized algorithm reduces the slice from 165K to 549 source line statements without degrading the accuracy (see case study II). Column (fun) reveals that dual slices are effective in capturing complex causal relations, including those that span multiple function boundaries.

5.1 Case Studies

Case I. Consider the following bug from the MySQL 5.0 (bug `m-3`):

```
1. CREATE TABLE a1 (id INTEGER NOT NULL\
            PRIMARY KEY AUTO\_INCREMENT);
   CREATE TABLE a2 SELECT id FROM a1;

2. DROP TABLE a1;
```

In MySQL, each client connection is handled by a separate thread. In this example, the above two `CREATE TABLE` queries are executed by one thread, T1, to create tables `a1` and `a2`, and the `DROP TABLE` query that drops `a1` is performed by another thread, T2. In most interleavings of T1 and T2, the program behaves normally: it either creates the tables and then drops `a1` or reports errors such as dropping a non-existent table or creating a new table from a non-existent table. However, as described in the bug report[2], if control is transferred from T1 to T2 at a specific program point where T1 is in the middle of creating `a2`, MySQL crashes. Even though the bug report describes how to reproduce the bug, it also states that it is very difficult to understand how the preemption leads to the failure.

Our technique computes a slice with only 23 source line instances. The dual slice and the traces for the two runs are shown in Fig. 7. The source code locations are also presented on the left. Note that these statement instances cross multiple functions and source code files.

In the failing run, steps (1) and (2) correspond to the initializations in T1. Step (3) occurs after T1 creates `a2` and is about to initialize `a2` with `a1`; it sets the `in_use` flag of `a1` to express its intention. Then, T1 is preempted by T2 after step (3) and then step (4) is executed, which tests the flag `in_use`. Steps (5) and (6) are executed due to this faulty branch outcome at (4). Step (5) tests if T2 is the one that sets `in_use` and it fails. Subsequently, T2 in step (6) sets the flag `some_table_deleted` to indicate it wants to delete a table and yields the control back to T1. T1 proceeds. At step (7), T1 tests if another thread has expressed its intention to remove a table. If so, T1 closes the table at step (8) and resets the `info` pointer at step (9), which results in the segmentation fault at (10) when it tries to use `info` to finish constructing `a2`.

In contrast, in the passing run, T2 executes after T1 finishes so that (4) is executed after T1 resets `in_use` at step (11) at the end of its execution, denoting it finishes using `a1`. Note that in the T1 execution, the fall-through path is taken at (7) and hence `info` is not null at (10).

Our dual slice precisely identifes the causal path of the failure. Translated to text, the generated slice can be read to mean "`in_use` *should not have had the value of* `thd` *(as manifested in the failing run) but rather the value of* `0` *(as defined in the passing run) at (4); the false branch of (4) should not have been taken, leading to the executions of (5) and (6);* `some_tables_deleted` *should not have the value 1, and hence* `info` *should not be 0, leading to the final failure*". Although not shown in the figure, accesses to `in_use` are protected with a common lock such that the root cause is an atomicity violation in the scenario of creating a table. Note that, the positive information from the passing run clearly pinpoints the root cause on `in_use`, which is not feasible otherwise.

Although the violation can be identified, the fault is hard to identify. Setting pointer `info` to null causes the crash, but the code for closing a table and cleaning up the pointer is not buggy. In many scenarios, step (4) receiving its value from step (3) (as part of the root violation in the failing run) is legal as it is part of the synchronization protocol. The goal of the complex protocol is to achieve high concurrency by not requiring queries to be executed atomically. The bug lies in a hole in the protocol that fails to consider this specific interleaving. Five of the bugs studied exhibit similar characteristics: they have intentional races and/or atomicity violations that by themselves do not contribute to the failure.

Case II. Fig. 8 presents the dual slice for the `mysql-5` bug. [3] It occurs because of an execution omission error that fails to properly clean up a pointer. In this case the server crashes on step (16) because `ref_item` is corrupted. Hence, we compute the dual slice on this variable at (16). Step (16) is control dependent on (15) and is data dependent on (14). Both (15) and (14) are value differences. The value difference at (15) causes the flow difference at (16). In the passing run (14) is data dependent on (12) while in the failing run it is data dependent on (13). Both are control dependent on step (11) that is a value difference. Slicing in the failing run has to stop at (11) because its dependence (step (3)) is not a trace difference.

Switching to slicing the passing run on (11) identifies this as an omission case because (10) was not executed in the failing run, resulting in a dangling pointer. Step (10) is transitively guarded by the predicate at (8), which is a value difference. The edge from (10) to (8) denotes a control dependence chain of length 45 because (10) is nested very deep in the region of (8). We indeed have each individual step in the chain captured in the slice, which explains the large slice size for this case. Suppressing such single line chains leads to much smaller slices with a corresponding loss of information. For instance, from the line numbers on the left, we can see (8) and (10) are very distant in the code and it is unclear why (10) is control dependent on (8) without the chain, which clearly provides the context under which (10) is executed.

[2] http://bugs.mysql.com/bug.php?id=12845

[3] http://bugs.mysql.com/bug.php?id=42419

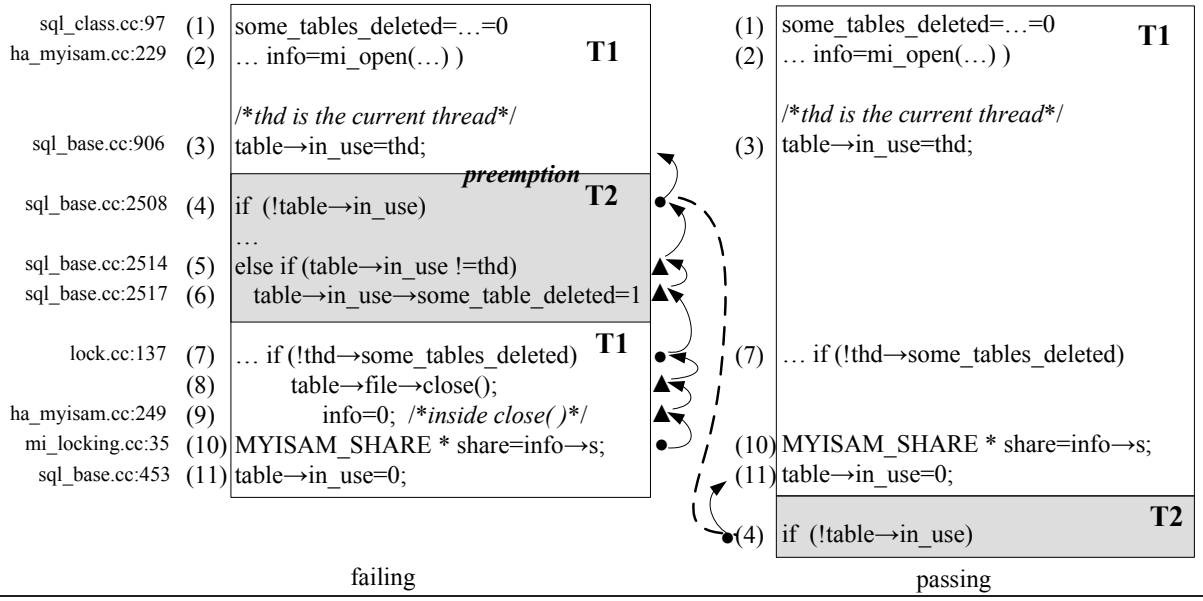


Figure 7. Case Study I.

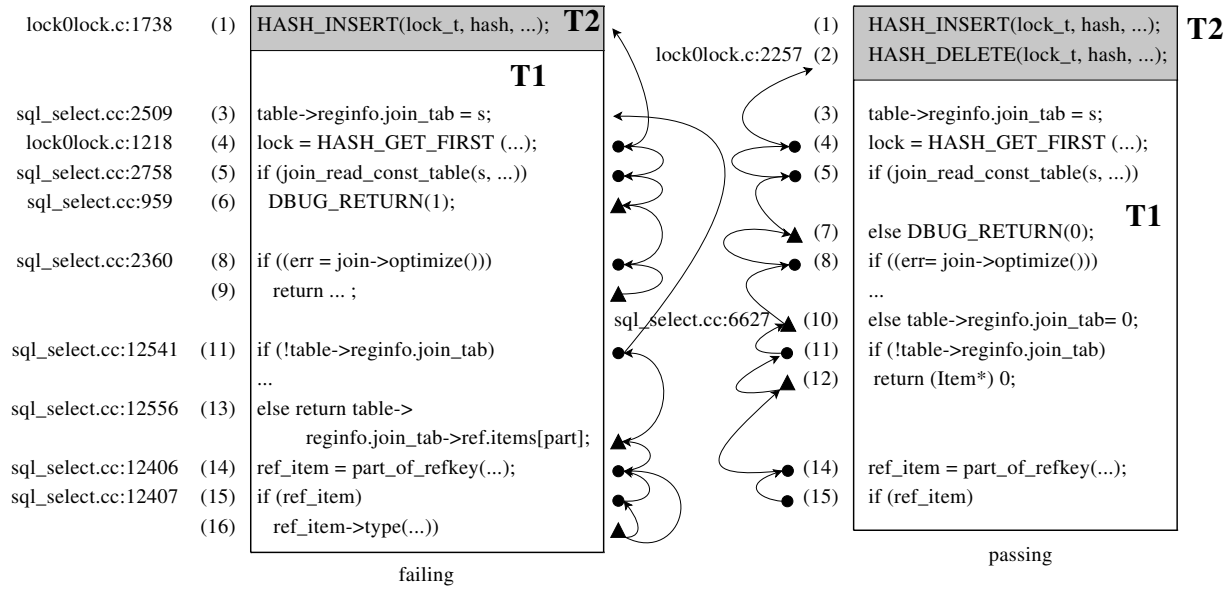


Figure 8. Case Study II.

Identifying the value difference (8), the algorithm can proceed. In the failing run `join->optimize()` returns 1 at (6) whereas it returns 0 at line (7) in the passing run, determined by the value difference on (5). Again the edges here represent transitive control dependence chains. At (5), `join_read_const_table` returns an error in the failing run indicating the server tries to abort a transaction. The error code is first detected when it tries to acquire a lock at (4) and fails. The root cause is identified by step (4) receiving the lock value from different places. It is an atomicity violation because (1) and (2) should be atomic. Note that step (1) and (4) are equivalent to a lock acquire while (2) is equivalent to a lock release.

6. Related Work

There has been extensive investigation of race detection [2, 14, 17, 18, 27] and atomicity violation detection [5, 7, 15, 22, 25]. When a race or atomicity violation is reported, however, it remains the programmer's responsibility to decide if the candidate is a false positive and, more importantly, to reason about how true positives lead to failures. As shown in this paper, the explanation of how a race induces a failure may oftentimes entail a complex causal chain of effects, not easily derived by mere identification of the race.

There has also been recent progress in devising techniques that can generate deterministic failure-inducing schedules [3, 4, 13, 16, 18]. These techniques systematically explore a bounded space of schedules with various search strategies. Our work is complemen-

tary to these techniques insofar as it can derive a meaningful explanation as to why a schedule lead to a failure.

In [28], a technique is proposed to compare a failing execution and a similar run to explain a failure. Such techniques rely on online memory state comparison and minimization during execution. As a result, they are often able to produce very concise failure causal chains. However, because the approach relies on mutating a passing run to a failing run by changing memory state, using the mutation as an indicator of failure relevance, it is unclear how it could be adapted to work in the presence of multiple threads.

There has also been work on slicing concurrent programs [8] and concurrent executions [20]. Our technique is more related to slicing execution. Existing solutions focus on the failing run. Without the passing run, they have to speculate what could have happened in a different run by considering write-after-read and write-after-write dependences, which are often numerous, resulting in slices larger than dynamic slices, which are already substantial as shown by our experimental results. Furthermore, these techniques are not robust in the presence of execution omission errors.

Recently, thin slicing [19] was proposed to selectively traverse dependences to produce very thin slices. Inspired by their observation, we leverage two runs and use trace differences to guide selective traversal to produce concise and precise failure explanations.

Solutions have been proposed for execution omission in the context of dynamic slicing [6, 29]. These solutions produce slices larger than dynamic slices. For example, in [29], a large number of re-executions are needed to reason about omitted dependences.

Cooperative Crug Isolation (CCI)[21], is a low-overhead instrumentation technique to isolate the root causes of concurrency bugs (or crugs). CCI inserts instrumentation that records occurrences of specific thread interleavings at run-time. This technique is complementary to our contributions, which provides a causal explanation for *why* a concurrency error occurs by comparing the behavior of passing and failing runs.

7. Conclusion

We have proposed a new dynamic analysis that identifies the root cause of different kinds of concurrency bugs. It does so by producing a causal path leading from the root cause to the failure point. Provided with two schedules, one corresponding to a correct run, and the other a failure, the technique collects traces for the two runs. It aligns the two traces and then identifies trace differences. Our dual slicing algorithm is applied to causally connect these trace differences. We show that the technique is highly effective in producing small yet accurate failure traces for real bugs in large concurrent software.

References

[1] Thomas Ball and James R. Larus. Efficient Path Profiling. In *MICRO'96*, pages 46–57.

[2] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *ISSTA'08*.

[3] F. Chen, T. F. Serbanuta, and G. Rosu. JPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE'08*.

[4] J. D. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules. In *ISSTA'02*.

[5] C. Flanagan and S. N Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL'04*.

[6] T. Gyimothy, A. Beszedes, and I. Forgacs. An efficient relevant slicing method for debugging. In *FSE'99*.

[7] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE'08*.

[8] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-Threaded Programs with JVM Concurrency Primitives. In *SAS'99*.

[9] P. Joshi, C.-S. Park, K. Sen, M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *PLDI'09*.

[10] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 1988.

[11] J. R. Larus. Whole Program Paths. In *PLDI'99*.

[12] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics. In *ASPLOS'08*.

[13] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *PLDI'08*.

[14] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP'03*.

[15] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *FSE'08*.

[16] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *ASPLOS*, 2009.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comp. Sys.*, 1997.

[18] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI'08*.

[19] M. Sridharan, S. J. Fink, and R. Bodik. Thin Slicing. In *PLDI'07*.

[20] S. Tallam, C. Tian, and R. Gupta. Dynamic Slicing of Multithreaded Programs for Race Detection. In *ICSM'08*.

[21] A. Thakur, R. Sen, B. Liblit and S. Lu. Cooperative Crug Isolation. In *WODA'09*.

[22] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *PPoPP'06*.

[23] W. Weimer, T. Nguyen, C. L. Goues, and S. Forres. Automatically finding patches using genetic programming. In *ICSE'09*.

[24] B. Xin, N. Sumner, and X. Zhang. Efficient Program Execution Indexing. In *PLDI'08*.

[25] M. Xu, R. Bodik, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *PLDI'05*.

[26] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA'09*.

[27] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *SOSP'05*.

[28] A. Zeller. Isolating Cause-Effect Chains from Computer Programs. In *FSE'02*.

[29] X. Zhang, S. Tallam, N. Gupta, and R. Gupta. Towards Locating Execution Omission Errors. In *PLDI'07*.