Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction

Dasarath Weeratunge Xiangyu Zhang Suresh Jagannathan

Department of Computer Science,
Purdue University
{dweeratu,xyzhang,suresh}@cs.purdue.edu

Abstract

Debugging concurrent programs is difficult. This is primarily because the inherent non-determinism that arises because of scheduler interleavings makes it hard to easily reproduce bugs that may manifest only under certain interleavings. The problem is exacerbated in multi-core environments where there are multiple schedulers, one for each core. In this paper, we propose a reproduction technique for concurrent programs that execute on multi-core platforms. Our technique performs a lightweight analysis of a failing execution that occurs in a multi-core environment, and uses the result of the analysis to enable reproduction of the bug in a single-core system, under the control of a deterministic scheduler.

More specifically, our approach automatically identifies the execution point in the re-execution that corresponds to the failure point. It does so by analyzing the failure core dump and leveraging a technique called *execution indexing* that identifies a related point in the re-execution. By generating a core dump at this point, and comparing the differences betwen the two dumps, we are able to guide a search algorithm to efficiently generate a failure inducing schedule. Our experiments show that our technique is highly effective and has reasonable overhead.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Debuggers; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Dumps, Tracing

General Terms Algorithms, Verification

Keywords concurrency bugs, reproduction, execution indexing, multi-core, Heisenbugs

1. Introduction

Much of the complexity in debugging concurrent programs stems from non-determinism that arises from scheduler interleavings (in single core environments) and true parallel evaluation (in multi-core settings); these interleavings are often difficult to precisely reproduce when debugging an erroneous program. Concurrency errors that occur under certain interleavings, but which are absent under others, are called Heisenbugs.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

In single-core environments, Heisenbugs can sometimes be repaired by recording all scheduler decisions [7] such that the failed execution can be faithfully replayed and examined. Unfortunately, this approach does not easily generalize to parallel multi-core environments where there are multiple schedulers, one for each core. When two operations executed in parallel on different cores access shared state, it is necessary to record the order in which these actions are performed; simply recording the thread schedule on each core does not provide this level of detail. Instruction-level monitoring that records the order in which accesses to shared variables occur [4, 8, 28, 18, 16] is expensive, however, and often requires hardware support, limiting its applicability.

There has been significant recent progress in testing concurrent programs on single-core systems that perform a (user-specified) bounded search of possible interleavings. Systems such as CHESS [17], CTrigger [22] and randomization techniques [27] leverage specific characteristics of concurrency-related failures to guide this exploration. For example, CHESS is predicated on the assumption that many concurrency failures can often be induced by injecting a few preemptions; CTrigger is based on the assumption that errors in concurrent programs often arise because well-defined access patterns for shared variables are violated. Because these built-in assumptions permeate their design, they behave quite differently from model-checking approaches which attempt to explore the entire space of interleavings,and face substantial scalability problems as a result. Regardless of the specific technique, existing approaches are not geared towards *reproducing* concurrency bugs in multi-core environments since they operate with no *a priori* knowledge of specific failures. Their generality, while useful for discovering new bugs, is less beneficial for reproducing known ones.

This paper targets the problem of reproducing Heisenbugs in parallel environments. Our technique combines lightweight analysis of a failure core dump with a directed search algorithm that uses the results of the analysis to construct a schedule that can reproduce the bug in a single-core environment. Notably, our technique does not assume low level logging or hardware support in the failing run, as long as core dumps can be generated when a failure is encountered, and the failure inducing program inputs are available. It requires very little program instrumentation and involves no modifications to the underlying thread scheduler. Indeed, programs can run on multiple cores with real concurrency in a completely normal fashion.

Our approach assumes core dumps will be generated when programs fail. Core dumps are expected to contain a complete snapshot of the program state at the point of the failure, including register values, the current calling context, the virtual address space, and so on. Given a core dump, our technique reverse engineers a

highly precise identification of the failure point that provides substantially greater precision than what can be derived using just the program counter (PC) and calling context. We leverage a technique called *execution indexing*(EI) [29]. EI generates a canonical and unique identification of an execution point, called the *index*, which can be used to locate the corresponding point in *other executions*. We present an algorithm to reverse engineer the index of the failure point from a core dump.

In the reproduction phase, the program is executed with the same input on a single core under a deterministic scheduler. Since our technique is geared towards Heisenbugs which by their nature are expected to occur rarely, it is very likely that the single-core run does not reproduce the failure. However, using index information gleaned from the core dump, our technique can locate the point in the passing (re-executed) run that corresponds most closely to the failure point. A core dump is generated at this point and compared to the failure core dump. The difference between the two core dumps, especially with respect to shared variables, reveals a wealth of information regarding salient differences between them. We enhance the CHESS algorithm to leverage this information to efficiently construct a failure inducing schedule from this point.

Our contributions are summarized as follows.

- We propose a novel concurrency failure reproduction technique that has negligible burden on concurrent program execution in multi-core environments. Our technique takes a failure core dump and generates a failure inducing schedule.
- We re-execute the program on a single core; we refer to this re-execution as the *passing run*. We leverage EI to pinpoint the execution point in the passing run that corresponds to the failure point in the failing run. We propose an algorithm to reverse engineer the failure index from the core dump. We also devise an algorithm that identifies the corresponding failure point in the passing run.
- We propose to generate a core dump in the corresponding point in the passing run and compare it to the failure core dump. We study two strategies to prioritize important value differences. One is based on temporal distance to the failure and the other is based on program dependences.
- We propose an algorithm based on CHESS that leverages value difference information to quickly find a failure inducing schedule.
- We conduct experiments to evaluate the cost and efficacy of our technique. The results on `mysql`, `apache`, and splash-II programs show that our technique entails 1.6% overhead on production runs. The experiment on a set of real concurrency bugs on `mysql` and `apache` demonstrates that our technique achieves orders of magnitude reduction on the number of schedules needed to be explored and on the time required to explore them, incurring only modest cost during the reproduction phase.

2. Overview

Consider the example in Fig. 1. Suppose two distinct threads execute functions `T1()` and `T2()`, resp. Variable `x` and array `a` are shared, pointer `p` is local to `T1`. Depending on the value of `a[i]`, pointer `p` may be set to 0 at line 8; `x` is used as a flag to indicate if `p` is a null pointer. The de-reference inside function `F()` is guarded by `!x` at line 11. The problem with this program is that the write to `x` at 7 and the read at 11 are not atomic; thus, there is a race between the read of `x` at line 11 and the write performed by `T2` at line 21. One possible fix is to enlarge the atomic region guarded by `lock` to include 11 and use the same lock to guard the write to `x` in `T2`.

A failing execution is shown in Fig. 2. Suppose the two threads execute in parallel. In the execution shown in (a), the loop executed

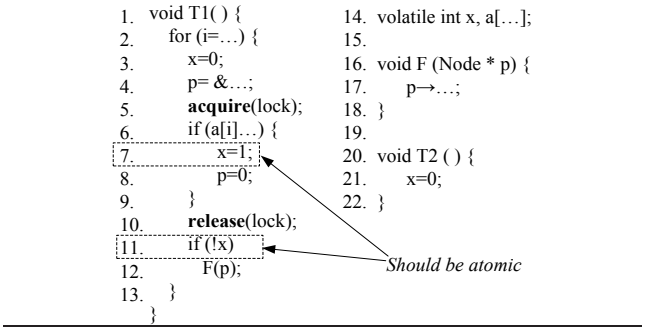


Figure 1. Example Code.

by `T1` iterates twice. In the second iteration, `p` is set to 0 and `x` is set to 1 at Ⓐ. However, `x` is undesirably reset at Ⓑ, resulting in the predicate at Ⓒ taking the true branch and eventually causing a null pointer dereference during the evaluation of `F` at line 12. When the failure occurs, a core dump is generated that records the current execution context, register values, and the contents of memory.

In the debugging phase, we re-execute the same program with the same input. Our goal in this phase is not to reproduce the error, but to construct a passing run under the control of a deterministic scheduler. The execution shown in Fig. 2 (b) results in `T2` being scheduled after `T1`; this in turn leads to the predicate at Ⓕ evaluating to `false`, ensuring the program completes correctly. Given the passing run, we can now compare its state with the state of the failing run to reproduce the failure.

Our technique consists of three steps. In the first step, it analyzes the core dump to uniquely identify the execution point where the program crashed (point Ⓓ) and tries to locate the same (closest) point in the passing run. In our example, the same point does not occur in the passing run. The closest (temporal) point is Ⓕ.

Note that using calling contexts as an abstraction of these program points is not very accurate. Assume in the first iteration of the loop in `T1` shown in Fig. 2(a), the predicate on `a[i]` takes the false branch so that `x` has the value of 0, leading to the predicate at 11 taking the true branch, resulting in the pointer being de-referenced inside `F()`. When the pointer is de-referenced, the calling context is the same as the context in the second iteration that results in the failure, namely `main` → `T1` → `F`.

To gain greater precision, we leverage a canonical execution point representation called *execution indexing* [29]. An execution point is uniquely represented by its *index*. In this paper, we devise an algorithm to reverse engineer the index of the failure point from the failure core dump. The failure index is used to find the corresponding point or the closest corresponding point in the passing run. Such a point is called *the aligned point* in this paper. In our example, since the predicate at Ⓕ does not take the true branch, it serves the role of the aligned point since it is the point closest to the failure point in the erroneous run.

In the second step, a core dump is generated at the aligned point (in the passing run), here Ⓕ. The core dump is compared to the failure core dump to identify the variables, particularly shared variables, that have different values across the two runs. These value differences are the result of schedule differences. In our example, the salient value difference is on `x`, as highlighted in the two core dumps.

In the third step, a schedule permutation algorithm in the spirit of CHESS [17] is used to permute the schedule in the passing run with the goal of inducing the failure. As the passing run completes successfully, the standard CHESS algorithm would try to generate preemptions at all synchronization points in the passing run, of which there may be many. In comparison, with the identification

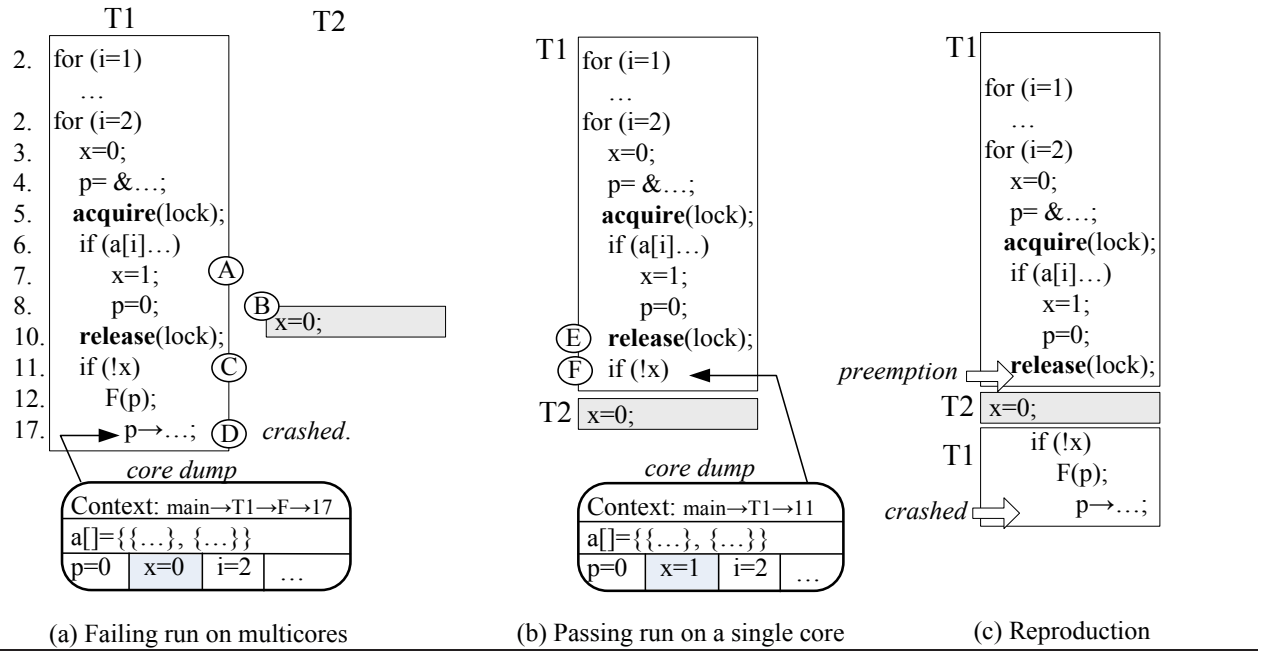


Figure 2. Overview. Plain boxes represent executions of `T1`; shaded boxes represent executions of `T2`; rounded boxes represent core dumps;

of the aligned point and the core dump analysis, our algorithm can focus on the set of synchronizations close to the aligned point, and it can selectively inject preemptions at those points likely to manifest the observed value differences, substantially reducing the search space that must be explored. In our example, there are potentially five preemption points in `T1`, corresponding to the beginning of `T1` and the `acquire` and `release` operations in the two iterations of the loop. Our algorithm concludes that the `release(lock)` operation at Ⓔ is the closest synchronization point that influences `x`. Consequently, the scheduler is instrumented to inject a preemption right after the lock release. By doing so, the failure is successfully reproduced.

3. Reverse Engineering Precise Failure Points

Identifying the execution point in a passing run that corresponds to the failure point, i.e., the aligned point, serves two critical goals: first, it locates the set of synchronizations that are close to the aligned point; second, it helps identify the salient variables that have faulty values by comparing the core dump at the aligned point and the failure core dump. To do so requires reverse engineering the precise identification of a failure point from a core dump.

Using the program counter (PC) of the failure point is the most straightforward way to identify the failure point. However, the instruction denoted by the same PC may be encountered multiple times during an execution; for example, it may appear in different calling contexts, or on different iterations of a loop. A more sophisticated approach is to use the calling context and the PC of the failure point as a signature. However, this is also not sufficient as exemplified by our earlier example in Section 2, in which the call to `F()` resides in a loop in which multiple execution points corresponding to different iterations all have the same calling context and PC. Theoretically, a program has a finite number of calling contexts but its execution may have infinite number of dynamic points, which implies that many execution points may alias to the same calling context and PC signature. An empirical study described in [6] confirms this hypothesis empirically, and shows that executions that produce billions of dynamic points may have less than one thousand unique calling contexts.

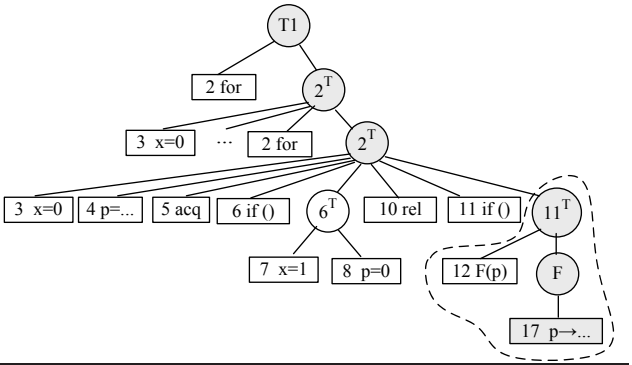


Figure 3. The Index Tree of the Execution of `T1` in Fig. 2 (a).

3.1 Execution Indexing

Execution indexing (EI) is a technique proposed in [29]. Execution points are uniquely represented by a signature derived from a program's dynamic control flow. Points across multiple executions are aligned by their indices – namely, two points are considered to be aligned if they have the same index.

The basic idea of EI is to use execution structure to correlate points across executions. The tree in Fig. 3, called an *index tree*, represents the structure of the thread executing `T1` in the failing execution in Fig. 2 (a). Leaf nodes are boxed, representing statement executions. Internal nodes are circled, representing the body of complex statement executions. Sample complex statements are conditionals and method invocations. The labels of the internal nodes represent the complex statement and the branch taken if applicable. In particular, the root node represents the entire thread. It consists of the loop statement 2. Since 2 is a complex statement and the true branch was taken, the execution within the true branch is represented by a node with the label 2^T. Note that because it is

a child of the root node, it is represented as being directly nested within the body of `T1`. Since statements 3, 4, 5, and so on directly nest in the true branch of 2, they are the children of the 2^T node. Furthermore, as the first execution of 2 takes the true branch, there is another iteration of the loop, leading to another 2^T appearing as the child node of the 2^T node on the second level. The process continues in this vein leading to the structure as shown in Fig. 3.

The structure of `T1` in the passing execution in Fig. 2 (b) can be similarly constructed. The primary difference lies in the fact that the predicate at line 11 in the second iteration takes a different branch. Hence, the two trees only differ at the isolated area at the right corner of the tree shown in Fig. 3. The two executions are aligned by aligning the trees. Intuitively, this structural alignment tolerates cases in which a predicate takes different branches across runs by aligning the execution points before and after the different branches.

At runtime, the index trees are usually not explicitly constructed. Instead, the *index* of the current execution point, which is *the path from the root of the index tree to the leaf representing the point*, is maintained. It represents the nesting structure of the current point. The index can be used to identify the aligned point in a different execution. For example, the index of the crash point in Fig. 2 (a) is the shaded path shown in Fig. 3. This index can be used to see if the same point is encountered in the passing run in Fig. 2 (b). In our example, the crash point is not executed in the passing run, reflected by fact that the corresponding index is not encountered. In contrast, the closest point in the passing run is the predicate instance denoted by the index of $\texttt{T1} \rightarrow 2^T \rightarrow 2^T \rightarrow 11$.

In order to maintain the current index, the current (transitive) nesting structure needs to be maintained. In other words, the branches and method bodies that the current execution point nests in need to be decided. To do so, two types of execution regions are defined. The first type of region concerns predicate branches and the second type concerns method bodies. Regions follow the last-in-first-out rule, meaning the last entered region must be exited first. Hence, a stack (an *index stack* (IS)) can be used to maintain the index of the current execution point. An entry is pushed to the stack if a region is entered. It is popped when the region is exited. The state of the stack reflects the nesting structure of the current execution point and can be used to construct the current index.

An online algorithm that computes nesting structure based on post-dominance analysis was proposed to deal with control flow caused by `break`, `continue`, etc., which violates syntactic constraints. More specifically, *a predicate branch region is delimited by the predicate and its immediate post-dominator*. In fact, all statement executions in a predicate branch region are control dependent on the predicate. Intuitively, a statement x is control dependent on the true/false branch of a predicate y if x's execution is directly determined by y taking the true/false branch [9]. *A method body region is delimited by the entry to the method and the exit from the method*.

Rule	Event	Instrumentation
(1)	Enter procedure X	IS.push(X)
(2)	Exit procedure X	IS.pop()
(3)	Predicate at p with the branch outcome being b	IS.push(p^b)
(4)	Statement s	while (p^b=IS.top() $\wedge$ s is the immediate post-dominator of p) IS.pop()

*IS is the indexing stack.

Figure 4. EI rules.

The instrumentation rules for EI are presented in Fig. 4. The first two rules mark the start and the end of a procedure by pushing and popping the entry associated with the procedure, respectively.

benchmark	one CD	aggr. to one	not aggr.	loop	total
apache-2.0.46	84.42	4.93	4.18	6.47	105K
mysql-5.1.31	89.92	2.77	3.1	4.22	892K
postgresql-8.3	86.46	3.4	2.7	7.44	521K

Table 1. The distribution of control dependences. Column "one CD" means the percentage of statements that have a single control dependence; column "aggr. to one" means although the statement has multiple control dependences, these control dependences can be aggregated to one; column "not aggr." indicates the number of statements that have multiple non-aggregatable control dependences; column "loop" are loop predicates. Note that these control dependences are all intra-procedural. Interprocedural dependences caused by function invocations are captured by the call stack.

In Rule (3), if a predicate is encountered, an entry comprised of the predicate and the branch outcome is pushed to the stack. Note that the branch outcome label is used to distinguish which of the two regions is entered. Finally, Rule (4) specifies that if the current executing statement is the immediate post-dominator of the top entry on the stack, the top entry is popped. The while loop is to handle multiple entries having the same immediate post-dominator. The state of the IS and the label of the current executing statement constitute the current index.

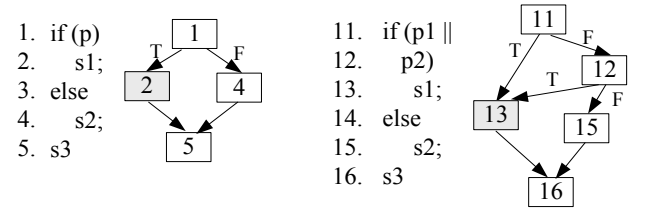
Consider the failing execution in Fig. 2 (a). When thread `T1` is spawned, an entry with label `T1` is pushed, which is only popped when the thread terminates. When the loop enters its first iteration, namely, predicate 2 takes the true branch, an entry with label 2^T is pushed. The entry will be popped when its immediate post-dominator, the end of the method, is encountered. Entering the second iteration results in another index, 2^T, being pushed onto the stack. Upon the execution of statement 3 in the second iteration, the concatenation of the current stack, [`T1`, 2^T, 2^T], and statement 3, precisely represents the index of the statement execution.

EI has been successfully used to associate corresponding points across multiple concurrent executions in the context of data race detection [29] and dead lock detection [13].

3.2 Reverse Engineering a Failure Index

As described earlier, maintaining EI requires instrumentation and thus runtime overhead. In [29], a highly optimized EI implementation entails 42% overhead on average, which is clearly too high to be used for production runs. Furthermore, such high overhead perturbs concurrent executions significantly, which in turn may mask failures that would otherwise appear in normal runs.

In this paper, we propose to reverse engineer the index of the failure point from the core dump, entailing negligible overhead during production runs. The key observation is that given a PC, we can almost always reverse engineer its immediate nesting region, which is denoted by a predicate or a method entry. The nesting region of the predicate or the method entry can be recursively computed until the whole index is recovered.



(a) One-CD (b) Aggregatable to One-CD

Figure 5. Examples for non-loop control dependences. Control flow graphs are presented to the right of the code snippets. Shaded boxes denote the given PCs.

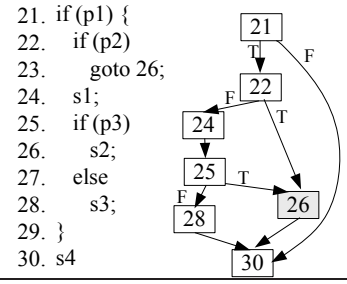


Figure 6. Example of a non-aggregatable non-loop control dependence.

Non-Loop-Predicate Statements with Control Dependences. We first consider the case in which the given PC is not a loop predicate and it nests in some predicate regions[1]. Through static control flow analysis, we can compute the static control dependences of the given PC, which denote the set of possible nesting regions at runtime. We observe that most statements have a single static control dependence, hence at runtime the given PC can only reside in one region. Fig. 5 (a) presents such an example. Assume at runtime, statement 2 is executed and we want to reverse engineer its nesting region. Since it is control dependent on one predicate, i.e. statement 1, it must reside in the true branch of statement 1. Hence, its parent node in the index tree must be 1^T. Table 1 presents the distribution of the various cases of control dependences in a set of concurrent programs. Observe that 84-89% of the statements have single control dependences.

It is also possible that a PC has multiple static control dependences. However, at runtime, it can have only one such dependence. Hence, we need to be able to reverse engineer the dynamic dependence from the multiple possibilities. Fig. 5 (b) presents an example. Statically, statement 13 is control dependent on 11^T and 12^T [2]. Dynamically, depending on whether the path $11 \rightarrow 13$ or $11 \rightarrow 12 \rightarrow 13$ is taken, 13 is control dependent on 11^T or 12^T, respectively. In other words, it may nest in the true branch of 11 or 12. As shown in Table 1, 2.8-5% of statements have multiple possible nesting regions caused by an OR operator. For such cases, we can aggregate the disjunction to one complex predicate so that 13 has only one nesting region. Let $11-12$ denote the complex predicate, the parent of 13 in the index tree can be reverse engineered as $11-12^T$.

In a more complex (and more unlikely) case, a statement may have multiple static control dependences caused by unconditional jumps. The multiple predicates can not be easily aggregated into a complex predicate. Our solution is to find the closest common single control dependence ancestor. In Fig. 6, statement 26 is statically control dependent on 22^T and 25^T. At runtime, it nests in one of these two regions, depending on the path taken. According to Table 1, 2.7-4.2% of statements fall into this category. In such a case, since both 22 and 26 are (transitively) control dependent on the true branch of 21, the parent node of 26 in the reverse engineered index is 21^T. We are losing some accuracy because we do not distinguish the two different paths leading from 21 to 26. However, we have not found this loss of precision to be a problem in practice.

Loop Predicates. If the given PC is a loop predicate (4.2-7.4% according to Table 1), its parent node in the index can be reverse engineered as well. We observe that loop related index subsequences are in the form of a string of consecutive loop predicates. For instance, consider the sample index tree shown in Fig. 3, the fact that the failure point is transitively nested in the second iteration of the for loop is represented by the failure index (the shaded path) having a substring of $2^T \rightarrow 2^T$, because the second loop predicate execution is dictated by the branch outcome of the first loop predicate execution. It is easy to infer that if the loop is exercised *n* times, in the *n*th iteration, the index stack will have a string of *n* consecutive loop predicates along the spine. If the loop has a loop count, its value can be easily recovered from the core dump. If the loop does not have a loop count, e.g., because it is generated via a while construct, our solution is to instrument the code to add a loop count. Since the instrumentation does nothing but increases the counter by one per iteration, the overhead is negligible. A detailed study of this approach is presented in Section 6.

Note that an execution index is not a full execution history but a precise indicator of an execution point. Hence, to reconstruct an index, it is sufficient to know the value of loop counters for the live loops (i.e., loops have not terminated) at the point of failure. These live loops are nesting, just like functions nesting in a calling context. The counters of loops that have terminated before the failure need not be maintained.

Statements Directly Nesting in Method Bodies. If the given PC does not directly nest in any predicate region, it must directly nest in the body of a method invocation. In such cases, the index parent node of the given PC is explicit from the call stack, which is an integral part of the core dump.

The algorithm is presented in Algorithm 1. Method **findParent()** is a recursive function that reverse engineers the index of a given PC from the failure core dump. To compute the index of the failure point, we invoke the method with the failure PC. Lines 2-6 handle cases that the PC directly nests in a method body. The call site and the nesting method is recovered from the calling context. The method is the parent node of the PC in the index. The algorithm proceeds with the call site PC. Lines 7-13 handle loop cases. The algorithm first retrieves the loop count value, *i*, and then inserts *i* entries of the loop predicate to the index. Lines 16-19 handle the PC having a non-loop predicate control dependence or multiple predicates that can be aggregated into a complex one. Lines 21-23 handle the non-aggregatable cases. At line 26, if the root of the index tree is reached, the recursive process terminates. Otherwise, it recursively calls itself to identify the parent node of the newly recovered index node.

Example. Consider our example in Fig. 2 (a). Method **findParent**() is called with the failure PC (line 17). This operation is not statically control dependent on any other statement. It thus directly nests in the method body of `F()`. The method is added as the parent of 17 in the index. Call site 12 is recovered from the call stack. Method **findParent**() is now recursively invoked with the call site PC, 12. Since 12 has a unique control dependence 11^T, node 11^T is added to the index. As 11 is control dependent on the loop predicate 2^T, the loop count `i=2` is retrieved from the core dump. Hence, entry 2^T is added twice to the index. Finally, the entry of thread `T1` is added and the process terminates. Observe that the index of the crash point is precisely reverse engineered.

Note that we only need to reverse engineer the failure index of the thread where the failure occurred. Specifically, we do *not* need to reverse engineer the indices of the current execution points of other threads. The reason is that schedule differences must have induced the failure through value differences in the failing thread.

[1] Switch-case statements are considered as falling into this category.

[2] According to [9], control dependence can be algorithmically determined as follows: *x* is control dependent on the true/false branch of *y* iff there is a path from *y* to *x* along the true/false edge of *y* such that *x* post-dominates each statement along the path except *y*.

Algorithm 1 Reverse Engineering Failure Index.

Input: the failure PC
Output: The index of failure PC, stored in *idx*
Definitions: *context* is the calling context, whose entries are in the format (*c*, *m*), meaning method *m* is invoked at call site *c*; **getLoopCount**(lp) retrieves the loop count value of loop lp from the core dump; p^b represents the *b* branch of predicate *p*.

/**find the index parent of a given PC, with respect to the failure core dump*/
findParent (*pc*)
1: *cd* = static control deps. of *pc*
2: **if** *cd*==ϕ **then**
3: /*directly nesting in the method body*/
4: (*callsite*, *method*)= *context*.pop()
5: *idx*= *method* • "→"• *idx*;
6: *parent*=*callsite*
7: **else if** *cd* contains a loop predicate lp^T **then**
8: /*directly nesting in a loop*/
9: *i*= **getLoopCount**(lp)
10: **for** *t* = 1 to *i* **do**
11: *idx*= lp^T • "→"• *idx*;
12: **end for**
13: *parent* = *lp*
14: **else**
15: /*directly nesting in non-loop predicates*/
16: **if** *cd*==$\{p^b\}$ or *cd* can be aggregated to p^b **then**
17: /*one CD or aggregatable to one CD*/
18: *idx*= p^b • "→" • *idx*
19: *parent*= *p*
20: **else**
21: q^b = the closest common CD ancestor of *cd*
22: *idx*= q^b • "→"• *idx*
23: *parent*= *q*
24: **end if**
25: **end if**
26: **if** *parent* $\neq$ the beginning of the thread **then**
27: **findParent**(*parent*)
28: **end if**

3.3 Identifying the Aligned Point in Passing Runs

With the recovered failure index, we can identify the point in passing runs that corresponds to the failure point. If such a point is not encountered due to schedule differences, we want to identify the closest alignment.

The proposed instrumentation rules are presented in Fig. 7. In passing runs, the failure index is provided in variable *idx*. The rules remove entries from *idx* when matching regions are encountered, until *idx* is empty, indicating the alignment of the failure point has been successfully identified. Rule (5) specifies that when a method is entered and it matches the head entry of *idx*, indicating the execution is about to enter a matching method body, the head of *idx* is simply removed. Rule (6) defines predicate instrumentation. If condition ① is satisfied, it means execution is about to enter a matching branch and hence the head of *idx* is removed. If condition ② is satisfied, meaning the same predicate is encountered but the branch outcome is different, the passing run is terminated with the `CLOSEST_ALIGNMENT` signal, meaning the exact alignment can not be found and this is the closest alignment. Intuitively, since the execution denoted by the remaining entries in *idx* must nest in the branch indicated by the head entry and the passing run is taking a different branch, it is impossible to match the remaining entries. It is worth mentioning that an important property of control dependence is that if *x* is control dependent on *y*, executing *y* implies executing *x*. If *idx* is a precise index, namely, an index strictly following the definition, when the current *idx* head entry *h* is removed, we know that the new head, denoted by h', must be executed, because h' is control dependent on *h* by definition. This guarantees our instrumentation rules can make progress.

However, we have only reverse engineered the failure index, which may miss some index entries because of non-aggregatable multiple static control dependences as described earlier. Condition ③ is defined to tolerate such inaccuracy. It specifies that if the current *idx* head entry is (transitively) control dependent on the opposite branch, which implies the execution will never reach the current head entry, the instrumentation also terminates with the `CLOSEST_ALIGNMENT` signal.

Rule (7) specifies that a successful alignment exists if the last entry of *idx* matches the currently executing statement, meaning all nesting regions have been successfully matched.

Rule	Event	Instrumentation
(5)	Enter procedure *X*	if (*idx.head*==*X*) *idx*-=*idx.head*
(6)	Predicate at *p* with the branch outcome *b*	if (*idx.head*==p^b①) *idx*-=*idx.head* elseif (*idx.head*==$p^{\neg b}$② || **controlDep**(*idx.head*, $p^{\neg b}$③) exit(CLOSEST_ALIGNMENT)
(7)	Statement *s*	if (|*idx*|==1 && *idx.head*==*s*) exit(EXACT_ALIGNMENT)

Figure 7. Instrumentation rules for finding the closest aligned point. Method **controlDep**(*x*, *y*) decides if *x* is transitively control dependent on *y*.

Example 1. Consider the failure index as shaded in Fig. 3. Assume it is provided to the instrumented passing run in Fig. 2 (b). Upon entering thread `T1`, the head node of the index, `T1`, is removed. Entering thread `T1` dictates that statement 2 must be executed, as the branch outcomes match, thus the first 2^T is removed. Similarly, the second 2^T is removed when the second iteration is entered. Upon the execution of 11 in the second iteration, since the branch outcome in the passing run is false when the index entry indicates true, we have according to rule (6) condition ②, a precise alignment mismatch, but have nonetheless found the closest alignment for the two executions.

Example 2. Consider the program in Fig. 6. Assume in the failing run, the path taken is $21^T \rightarrow 22^F \rightarrow 24 \rightarrow 25^T \rightarrow 26$ and the failure occurs at 26. As discussed earlier, due to the non-aggregatable multiple static control dependences of 26, the reverse engineered index is $21^T \rightarrow 26$. Assume in the passing run, the program takes the path $21^T \rightarrow 22^F \rightarrow 24 \rightarrow 25^F \rightarrow 28 \rightarrow$ `END`, that is, taking a different branch at 25. Upon executing 21 with the true branch outcome, the 21^T entry of the index is popped. Upon executing 25^F, since 26 is control dependent on 25^T, the condition ③ of Rule (6) applies and the instrumentation signals finding the closest alignment.

4. Identifying Critical Shared Variable Accesses

In the previous section, we introduced how to identify the aligned point in a passing run. Recall that the aligned point could be the exact alignment or the closest alignment. A core dump is generated at the aligned point. Critical shared variables are identified by comparing the core dump with the previously acquired failure core dump. Accesses to the critical shared variables are also identified and prioritized to drive schedule perturbation.

We consider a core dump to be a complete snapshot of the program state, including the call stack, registers, and the entire

virtual space. In other words, the current states of all active threads are captured. We compare the values of all global variables, the local variables on the current stack frame of the failing thread, and all the heap variables reachable from registers, global variables or the local variables of the failing thread. Note that it is not necessary to compare variables in all threads as the failure must be caused by some value differences *in the failing thread*. We use the algorithm in Boehm's garbage collector [5] to identify all reachable heap variables. The basic idea is to traverse memory regions through pointer fields as much as possible. We call the path leading from a register, a global pointer or a local stack pointer to a memory variable the *reference path* to the variable. We compare all the memory variables that are of primitive types, e.g. `char` and `int`, and which have identical reference paths in the two core dumps. Note that a memory variable may have multiple reference paths in the presence of aliasing. In this paper, we treat the aliased memory variable as multiple variables, identified by the different reference paths associated with it.

The core dump comparison produces a set of value differences. We focus on value differences of shared variables. The shared variables that have different values in the two core dumps are called critical shared variables (CSVs), because they reflect the outcome of schedule differences. They are also the reason why a failure occurs in one run but not the other. The schedule perturbation, as will be discussed in Section 5, is guided by the accesses to the CSVs in the passing run. More specifically, we want to perturb the benign CSV accesses to produce the failure. In this paper, we study two strategies to prioritize CSV accesses: *temporal distance* and *dependence distance*.

Prioritization Based on Temporal Distance. This heuristic prioritizes CSV accesses according to the temporal distance between the access and the aligned point. The intuition is that in the failing run, the CSV accesses critical to the failure are often close to the failure point. Since we do not monitor the failed run, we use the temporal distances to the aligned point in the passing run as an approximation. Moreover, since all passing runs are executed via a deterministic scheduler on a single core, we can easily identify all accesses that occur before the aligned point and only prioritize these accesses. For our example in Fig. 2 (b), `x` is the CSV, and the read of `x` at 11 in the second iteration is the closest access to the aligned point, the write `x=1` inside the predicate in the second iteration is the second closest, and so on. The write `x=0` in `T2` is not considered as it occurred after the aligned point and did not contribute to the value difference at the aligned point.

Prioritization Based on Dependence Distance. This heuristic prioritizes CSV accesses according to the dependence distance between an access and the aligned point. The intuition is that in the failing run, the CSV accesses critical to the failure must have contributed to the failure through data/control dependences and they tend to be close to the failure point along dependence edges. Since we do not have dependence information in the failing run, we use the dependence distance in the passing run as an approximation.

Specifically, we perform dynamic slicing [15] from the aligned point in the passing run with the variable that causes the behavior differences. If the exact alignment is identified, the variable that triggers the crash in the failing run is used as the slicing criterion. If only the closest alignment is identified, it must be the case that the two runs diverge at a predicate, and the variables that are used in the predicate are used as the slicing criteria. Note that these variables could be non-shared variables. The CSV accesses are ranked by their distances to the slicing criteria. Those that are not in the slice are given the lowest priority as they are very likely not relevant to the failure. In Fig. 2, since the passing run and the failing run differ at the predicate execution at line 11, the variable that caused the difference is used as the slicing criterion, namely, `x`. The most critical read to `x` at 11 is closest to the slicing criterion. The write `x=1` inside the predicate in the second iteration ranks the second. The same write in the first iteration has the lowest priority as it is not in the slice and hence not relevant to the failure. Note that the temporal distance heuristic can not exclude it.

5. Reproducing Failures

The last phase of our technique is to search for a failure inducing schedule with the guidance of CSV accesses. We enhance the CHESS [17] algorithm, which is used for testing concurrent programs, for this purpose. The idea of CHESS is to insert preemptions at synchronization points in a systematic way such that the space of interleavings can be algorithmically explored to find a failure inducing schedule. Even though the number of possible preemption combinations is exponential, the number of preemptions that must be used in combination with one another to trigger a failure is often bounded.

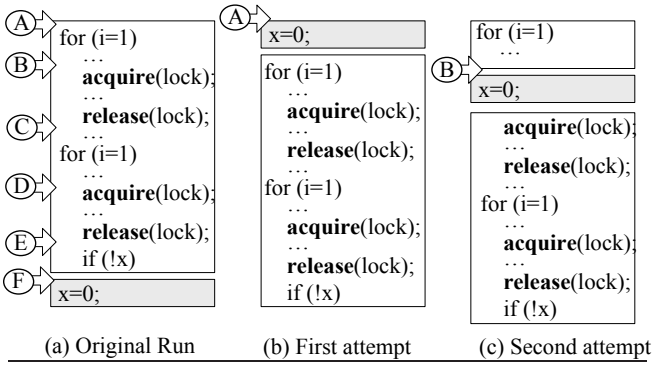


Figure 8. Applying CHESS to the passing run in Fig. 2 (b).

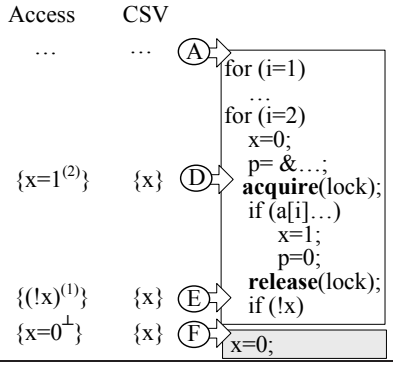


Figure 9. Enhancing CHESS. Each access is superscripted with its priority; symbol ⊥ represents the lowest priority.

We use the running example to illustrate the CHESS algorithm. Fig. 8 (a) shows the passing run in Fig. 2 (b). Labels Ⓐ to Ⓕ indicate possible preemption points. They are all associated with synchronization operations or the beginning of threads. Based on program semantics, preemption may be injected before or after a synchronization. For example, the preemptions associated with `acquire(lock)`, such as Ⓑ, are before the `acquire(lock)`, to allow threads needing `lock` to be scheduled. For a similar reason, preemptions associated with `release()` are after the lock release, e.g. Ⓒ. Given a set of candidate preemption points, the algorithm adopts several strategies to generate tests. The simplest strategy is linear search [17], namely, induce one preemption point

at a time in a linear fashion. Fig. 8 (b) and (c) show the first two preemption attempts. In the first attempt, `T1` is preempted such that `T2` runs before `T1`. In the second attempt, `T1` is preempted at Ⓑ such that `T2` is executed before `T1` acquires the lock.

Algorithm 2 Search for Failure Inducing Schedule.

Input: A list of preemption candidates in the execution order of the first passing run, stored in *preemption*.

Output: the preemptions that are needed to reproduce the failure.

Description: a preemption is a triple (*idx*, *accesses*, *csv*), with *idx* being the index that uniquely identifies the preemption point, *accesses* the CSV access annotation, *csv* the CSV annotation; *wl* a list containing weighted preemptions; *k* is the preemption bound; and, method **testrun(*s*)** applies a set of preemptions.

findSchedule(*preemption*)

```
1: for i=1 to k do
2:    for each i-subset of preemption, denoted as s, do
3:       w=Σ_{pm∈s} (the minimal priority superscript in pm.accesses)
4:       wl+= (w, s)
5:    end for
6: end for
7: sort wl in an ascending order of weight.
8: while wl ≠ ϕ && the failure is not reproduced do
9:    (w, s)= wl.pop().
10:   testrun(s)
11: end while
12: return s
```

preempt (*pm*)

```
21: for each thread T other than the preempted one do
22:    csv=the CSV set of the current synchronization point of T
23:    if ∃v ∈ csv, v is accessed by pm.accesses then
24:       create a check point and continue the execution with T
25:       if the failure is reproduced then
26:          exit()
27:       end if
28:       restore the check point
29:    end if
30: end for
```

Besides applying preemptions, the algorithm also controls the scheduler to systematically pick up the available threads to run. For instance, if there are threads `T3` and `T4` in our example, the algorithm also explores different schedules so that both can run at chosen preemption points.

CHESS is intended as a testing tool that explores all possible preemption combinations for a given bound. Because we have information regarding the source of a failure, we can direct the search space more profitably. Next, we present an enhanced CHESS algorithm that exploits information gleaned earlier.

We identify the sequence of preemption candidates from the passing run. We call the execution delimited by a preemption candidate *pm* and its immediate following preemption candidate the *schedule block* led by *pm*. Our scheduler never preempts a schedule block and hence all statement executions inside a block belong to the same thread. We annotate each preemption candidate with two pieces of information.

The first is the set of CSV accesses that are within the schedule block led by the preemption. Such information is used to prioritize the preemptions because it indicates what accesses may be perturbed if the preemption were triggered. The second piece is the set of CSVs that will be accessed by the current thread in the future. It is computed by aggregating all the CSVs that are accessed by the thread after the preemption. Such information is used to guide the scheduler to select threads when preemptions are applied.

For example, in Fig. 9, at the preemption candidate Ⓓ, the set of accesses is $\{\texttt{x=1}^{(2)}\}$, denoting that there is a CSV write `x=1` in the execution between Ⓓ and Ⓔ, and its priority is 2. The CSV set is $\{\texttt{x}\}$, denoting that `T1` will access `x` from Ⓓ to the end of the thread.

The search algorithm is presented in Algorithm 2. Assume a preemption bound of k.[3] Lines 1 and 2 generate preemption combinations that contain less than or equal to k preemptions. Each combination is assigned a weight w that is computed as the sum of the highest priority (the smallest superscript) of the accesses in each member preemption (line 3). For instance, assume a two preemption combination $\{pm_1, pm_2\}$, in which the accesses of pm_1 is $\{x = 1^a, y = 3^b\}$ and the accesses of pm_2 is $\{x = 0^c, y = 5^d\}$. Its weight is $min(a,b) + min(c,d)$. Combinations are inserted into the worklist *wl* at line 4. After all combinations are generated, the worklist is sorted in an ascending order at line 7. The loop in lines 8-11 applies each combination in the worklist in order until the failure is reproduced or the worklist is exhausted.

Method **preempt()** presents the scheduling algorithm when a preemption *pm* is applied. For each thread T other than the preempted one, the algorithm tests if *pm.accesses* has any overlap with the CSV set of the current synchronization point of T. Recall that *pm.accesses* contains the CSV accesses in the schedule block led by *pm* and the CSV set of T is the set of CSVs that will be accessed by T. Intuitively, the algorithm tests if switching to executing T may perturb the CSV accesses in the preempted schedule block. If so, the scheduler selects T to continue execution at line 24. All possible selections of T will be explored.

Example. Consider our running example. According to the preemption candidates and their annotations as shown in Fig. 9, the sorted worklist is $\{(1,\{Ⓔ\}),\ (2,\{Ⓓ\}),\ (3,\{Ⓔ,Ⓓ\}),\ ...\}$. When applying the first combination in the worklist, `T1` is preempted at Ⓔ, `T2` is the only thread that can be scheduled and its CSV set contains `x`. The accesses of Ⓔ is $\{\texttt{(!x)}^{(1)}\}$, in which `x` is accessed. According to the test at line 23, the search algorithm selects `T2` to execute next and thus reproduces the failure.

6. Evaluation

Our implementation consists of six components. The first is the static instrumentation engine that is responsible for instrumenting deployed software to add loop counters. To achieve maximum generality, we implement it on GCC-4.1.2. This is the only component that is expected to be used in the production environment. The remaining components are only used for reproduction in the debugging phase. The second component is the post-dominator and control dependence analysis. It is also implemented in C. The third component is for failure index reverse engineering and core dump comparison. The fourth component is a tracing system on Valgrind [19] that collects traces for slicing. It is also responsible for locating the aligned point when the failure index is given. The fifth component is the enhanced CHESS [17] algorithm. It is implemented on Valgrind. For comparison purpose, we have also implemented the original CHESS algorithm. The sixth component is the dynamic slicing algorithm mentioned in [30]. We implemented it with C. The experiments were conducted on a Intel Core 2 Duo 2.26GHz machine with 4GB memory, running Linux 2.6.

[3] For our experiments, we set $k = 2$ because it was shown in prior work [17] that most failures only need two preemptions to trigger.

Table 2. Concurrency Bugs Studied.

bugs	id	description	exec. time	threads
apache-1	21285	atom*	1.2s	3
apache-2	45605	race	1.4s	2
mysql-1	21587	atom	5.5s	2
mysql-2	12228	atom	4.9s	2
mysql-3	12212	race	1.5s	2
mysql-4	12848	atom	6.8s	2
mysql-5	42419	atom	14.2s	2

*`atom` means atomicity violation.

Table 3. Core Dump Analysis.

bugs	core dump (F+P)	vars/diffs	shared/CSV	len(index)
apache-1	108/108MB	23273/38	2600/5	49
apache-2	99/99M	2975/30	123/7	13
mysql-1	48/48MB	6686/64	1665/30	27
mysql-2	55/55MB	8310/359	2171/60	50
mysql-3	49/49MB	2294/118	840/11	51
mysql-4	45/45MB	4150/86	1877/71	35
mysql-5	158/158MB	17289 / 701	728/67	84

Table 4. Failure Inducing Schedule Production.

bug	chess*		chessX+dep		chessX+temporal	
	tries	time	tries	time	tries	time
apache-1	1028	18hr	832	14.6hr	644	10.9hr
apache-2**	63	2.2hr	34	4658s	27	3078s
mysql-1	760	18hr	4	3189s	4	3189s
mysql-2	421	18hr	5	1152s	5	1152s
mysql-3	712	18hr	7	940s	7	940s
mysql-4	619	18hr	6	3880s	6	3880s
mysql-5	562	18hr	6	3453s	6	3453s

*Executions were cut off after 18 hours if the bugs was not reproduced.
**The plain `chess` is able to reproduce the bug.

Table 5. ChessX+Temporal Using Instruction Count.

bugs	instrs.	vars/diffs	shared/CSV	chessX+temporal	
				tries	time (s)
apache-1	400M	22715/128	100/1	1329	24hr
apache-2*	112M	2975/33	116/10	54	1.9hr
mysql-1	7459M	6586/180	1576/48	50	6hr
mysql-2*	8954M	7209/163	2245/90	36	4.5hr
mysql-3	2708M	5583/229	1941/49	30	6hr
mysql-4	16285M	4104/203	1663/101	28	6hr
mysql-5	17456M	10711/383	1083/39	33	6hr

*the bugs are reproduced.

We select a set of bugs from `mysql` and `apache` to evaluate the effectiveness of our technique. These programs are multi-threaded and have been widely used as subjects for concurrency debugging. The bugs are on the full version of the programs. Since the bug repositories for these programs do not provide core dumps, we manually inspect the reports to extract the required input and environmental setup. Since the original inputs from the bug reports are usually very short, leading to only a few milliseconds of execution, we lengthen these inputs by prepending randomly generated inputs. We then instrument the programs to add necessary loop counters. We subsequently perform stress testing with the generated input on multiple cores to produce the reported failures. If the failure is exposed, we collect its core dump. Table 3 shows the set of failures that we successfully produced. The `id` column presents the bug ids in their repositories. The bug charateristics are described in the `description` column. The original execution time on multiple cores and the number of threads are presented in `exec. time` and `threads`, resp. It is worth mentioning that while stress testing is very expensive, it is not part of our proposed technique, but is used only to acquire the failure core dumps. After the core dump is collected, the program is executed with the same input on a single core under our Valgrind tracing component, which generates traces and a core dump at the aligned point. The two core dumps are then compared. The results are fed to our schedule search algorithm to produce the failure inducing schedule.

Table 3 presents the results of the core dump analysis. The `core dump` column presents the sizes of the core dumps. The `vars/diffs` column presents the number of variables that are reachable from the failing thread and hence subject to comparison, and the number of variables having different values in the two core dumps. Column `shared/CSV` presents the number of shared variables compared and the number of critical shared variables (CSVs), i.e. shared variables with different values. The last column presents the length of the reverse engineered failure indices. Since the passing run is performed inside Valgrind to locate the aligned point, the generated core dump also contains the state of Valgrind. To compare the core dump sizes, we exclude the part from Valgrind. We can observe that the failing and the passing core dumps have roughly the same size, indicating their memory mappings are roughly the same[4], as the consequence of generating the coredumps at the aligned points. Note that while many variables are reachable in the failing thread, very few of them have different values in the two core dumps. Also, the CSVs represents a small fraction of the total number of shared variables, indicating that CSVs can effectively reduce the schedule search space.

Table 4 quantifies the effectiveness of our technique. We denote the original CHESS algorithm, our enhanced algorithm with the temporal distance heuristic, and the enhanced algorithm with the dependence distance heuristic as `chess`, `chessX+temporal`, and `chessX+dep`, respectively. For each algorithm, we collect the number of schedules tried and the total time to execute the schedules. In most cases, our algorithm requires less than 10 tries while the original `chess` algorithm can not find the preemptions within 18 hours, even after a few hundreds tries. We believe these results support the claim that our technique is able to direct the search quickly to the failure. Our current implementation is on Valgrind, a relatively slow dynamic instrumentation system. Even without any instrumentation, Valgrind could slow down the original execution by a factor of 4-10. We have not yet attempted to perform any substantial optimizations to reduce this overhead. We also observe that `chessX+dep` is able to reduce the number of tries for two out of the seven cases.

To show the benefits of using execution indexing over simply using instruction counts to locate the failurep point, we also acquire the number of thread-local executed instructions from hardware counters when the failure occurs. In the passing run, we execute the same number of instructions for the same thread and then look for the execution of the failure PC. Such a point is considered as the aligned point. The rest of the procedure is same as our indexing based approach. Note that we do not consider a design that uses the instance count of the failure point PC because it entails significant overhead on production runs due to the cost of maintaining per-PC counters. The results are presented in Table 5. The `instrs.` column represents the thread local instruction counts when the failures occur. The next two columns present the core dump comparison results and the last two columns present the result of running our

[4] Core dumps are geneated by dumping the mapped memory segments.

Table 6. Other Cost.

bugs	core dump parsing time (s)	diff (s)	slicing (s)
apache-1	16	0.191	39.1
apache-2	7	0.003	30.3
mysql-1	343	0.025	33.9
mysql-2	331	0.066	41.1
mysql-3	299	0.030	35.7
mysql-4	190	0.048	32.3
mysql-5	728	0.200	45.8

chess algorithm, guided by the core dumps. We can observe that the number of reachable variables are quite different from those in Table 3 because of the different definitions of aligned points. The number of variable differences and CSVs is also different (notably, the number of CSVs is often larger than the corresponding number in Table 3 in many cases). The reason for this difference is that many of these variables are not frequently updated, making them insensitive to core dump timing. Finally, the important observation is that most failures (5 out of 7) can not be reproduced within a reasonable timeframe. This is because: (1) the set of CSVs are different, the real critical shared variables are not present in the CSV set; (2) the search algorithm starts at a wrong point (i.e. the aligned point according to instruction count) preventing the right pre-emption(s) from being located.

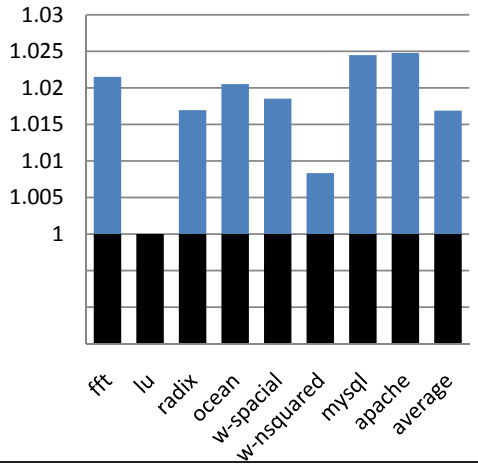


Figure 10. Runtime Overhead on Production Systems.

Fig. 10 presents the overhead of our loop counter instrumentation. Besides `mysql` and `apache`, we also select the concurrent programs from the splash-II benchmark as our subjects because they are more loop intensive[5]. In order to minimize the effect of non-determinism on instrumentation overhead, we run the benchmarks on a single core with a deterministic scheduler. For the results, we can see the overhead ranges from 0-2.5% with an average of 1.6%. This supports our claim of the technique has negligible runtime overhead on production runs. Note that although splash-II programs are loop intensive, many of their loops have loop counters and do not need to be instrumented, which explains why they have lower overhead than `apache` and `mysql`.

Table 6 quantifies other costs. Columns `core dump parsing` and `time to compare` present the times to parse and compare the two core dumps, respectively. It is clear that parsing core dumps is the dominant cost in core dump analysis. The reason is that the core dumps are very large and we currently use an expensive GDB coredump interface to retrieve variable values, which entails sending string queries to (and parsing string results from) GDB. We expect an online algorithm that does not rely on the GDB string interface to substantially reduce such costs. Another possible optimization is not to parse the entire core dump, but rather selected (relevant) portions. Column `slicing time` presents the slicing cost. Due to the length of the considered executions, full traces are too expensive to collect. We collect traces for a window of 20 million instruction executions, roughly 400MB. Recall that we perform dynamic slicing on traces. We find that these traces are sufficient to drive our algorithms. It is worth mentioning that these costs are all one time costs because they are only needed for the first re-execution.

[5] Some 32 bits splash-II programs are not included because our GCC instrumenter failed to compile them.

Case Study Next, we perform a case study on `apache` with the bug id 21285, i.e. `apache-1`. The apache web server maintains a cache shared by threads for processing requests. Content objects are placed into the cache in two steps. In the first step, an object is added to the cache with a default size, since at this stage the exact size of content is unknown. In the second step, when the exact size of the object is available, the object that was added earlier is first removed from the cache and then placed in the cache again with the proper size. This strategy allows early detection when multiple requests try to cache the same content. However, since the two steps are not atomic, an object with a default size could be evicted from the cache before it is replaced by an object with the proper size, leading to a crash as we explain below.

A part of the relevant code is presented in Fig. 11. To handle a request, a thread first calls `create_entity()` to place content with a default size in the cache. Subsequently, when the correct size is known, in `write_body()`, the same thread removes the content it added earlier, modifies its size and replaces it in the cache. Observe that the lock `sconf` → `lock` is not held across the two methods, leading to the possibility of an atomicity violation. When new content is inserted into the cache in `cache_insert()`, existing content is evicted if the projected total cache size exceeds the limit.

The configuration we used to trigger the bug, according to the bug report, is a cache that allows 2 objects, and 3 threads that handle three respective requests that demand caching. In the failure core dump, we observe that the program has crashed on **cache_cache.c**:182. The failure index is not encountered in the passing run. In fact, the two runs diverge at the predicate at line 181. When the failure core dump is compared against the core dump collected at 181 in the passing run, we find 5 CSVs out of the 2600 shared variables. The variable `c` → `current_size` used at 181 is one of the CSVs. In this study, we only inspect the results of using the dependence distance based strategy. Accesses to another CSV with the reference path of `cache_cache` → `pq` → `size` are also present in the slice. The variable is not shown in the code snippet for brevity. It keeps track of the number of objects in the cache.

With the ranked CVS access information, our algorithm tries 640 one-preemptions and 4 two-preemptions before it finds the failure inducing schedule. The generated schedule demands two preemptions: one at line 545 and the other at line 175, both are synchronizations leading schedule blocks that access `c` → `current_size`. The corresponding execution perturbation is as follows. The first thread is preempted just before it acquires the lock at line 545. Observe that at this point it has not placed any content into the cache. Then the second thread is run, and it is preempted at line 175. At this point, this thread has placed content into the cache with a default size, but the object has not been updated with the proper size. Now, the cache has one object. The third thread is run to completion and the number of objects in the cache

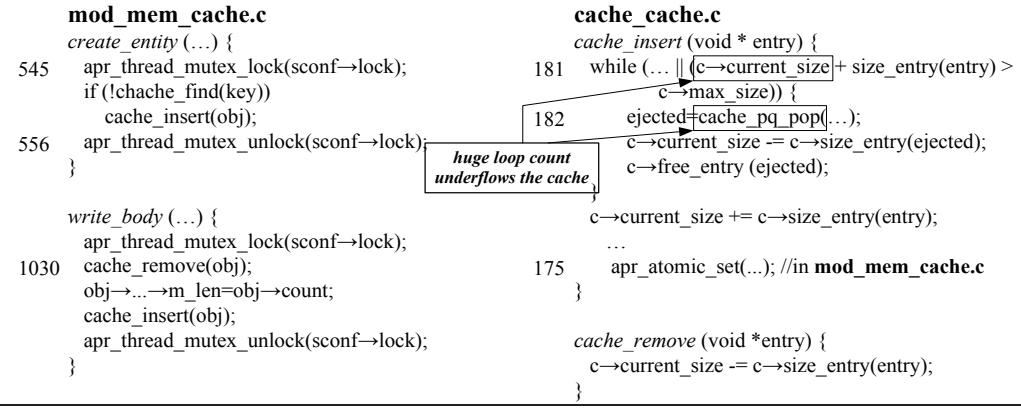


Figure 11. Case study of `apache-1`.

increases to two (the limit). Now of the two remaining threads, the scheduler picks the first thread to run as it comes before the second in the canonical order. When the first thread tries to place its content into the cache, the size limit is exceeded and the cache chooses to evict the object placed in cache by the second thread inside function `cache_insert()`. After the first thread completes, the second is resumed. However, when the second thread tries to remove the (already evicted) object from the cache at 1030, it ends up subtracting its size from $c \rightarrow$ `current_size` again. This leads to a negative number which manifests as a very large positive value since the field is an unsigned integer. Given this value, when the thread tries to place the content back into the cache, the huge loop count underflows the object queue at line 182.

7. Limitations and Discussion

Our technique currently assumes that the failure inducing input can be acquired and used in re-executions, which may not be true if the servers have been running for a long time. A potential solution is to use a lightweight checkpointing technique [25, 28, 23] to avoid the need to re-collect all inputs from the beginning of the execution. It would then only be necessary to reconstruct execution from the closest checkpoint and consider the inputs processed thereafter.

Our technique relies on core dumps. Some concurrency-related failures may not crash, but rather produce wrong outputs, although most bug reports we have seen for `mysql` and `apache` fall into the crash category. While core dumps can be acquired at wrong output points, we have not investigated the efficacy of our approach on non-crashing but erroneous executions.

To mitigate privacy concerns that may arise because of the need to supply coredumps on production runs, we note that our technique only requires sufficient information to identify shared variables that carry different values; the exact values of the shared variables are not important. Furthermore, recent techniques on anonymizing end-user information [10] to protect privacy apply naturally in our setting.

State drifting [1] describes scenarios in which concurrent schedules may quickly diverge significantly from sequential schedules. Though state drifting makes a vast array of states a possibility, it is not necessary to compare the closest correct state against the faulty state where the bug was observed. In practice, there exists a degree of freedom because both the states under which the bug could be reproduced and the possible candidates that could be used to compare against the faulty state increase as the total number of possible states increase.

There are other contexts that may give rise to concurrency bugs that we have not yet considered. For example, race conditions that arise due to relaxed memory consistency support in hardware [26], cannot be reproduced with a serial schedule. Moreover, our technique can not replay kernel and device state since it operates purely in user space. Hence, it does not handle bugs that are triggered by kernel actions. From our experience with the bug reports for the considered programs, such cases are rare. It is also possible that the different state of a CSV may have been overwritten by other writes before the core dump occurs. However, this is only problematic when the overwrites happen to make the variable have the same value in the two runs (so that it does not manifest itself as a CSV); we have yet to see such conditions in the bug reports we have examined.

8. Related Work

The prior work most relevant to ours is search-based reproduction techniques. In [23], a multi-phased reproduction technique is proposed. Specifically, coarse-grained logging is used in production runs to collect system call and synchronization information; while such coarse-grained information does not guarantee reproducing failures, a search algorithm is used to generate failure inducing schedules. In [2], a technique is proposed to search for executions based on output constraints, namely, constraints that produce the same erroneous output. Limited logging is needed in production runs to collect input traces, path profiles, and event orders to reduce the search space. A constraint solver is used to reproduce failures. Compared to these techniques, our approach shares the same observation that software-based approaches must perform directed schedule search because low overhead coarse-grained logging is not sufficient for faithful replay. The unique feature of our solution is that we reduce the search space by analyzing core-dumps, leveraging the idea of execution indexing. As a result, our technique has negligible overhead on production runs.

There are also software based replay systems that record individual memory accesses and their happens-before relations [4, 8]. Such systems entail substantial runtime overhead. There has been substantial work on software-based record and replay for applications such as parallel and distributed system debugging [21, 25, 24, 11, 3, 14, 20] . These systems only perform coarse-grained logging at the level of system calls or control flow and hence are not sufficient for reproducing concurrency failures. We consider these techniques complementary to ours.

Recently, it has been shown that with architectural support, concurrent execution can be faithfully replayed [12, 16, 18, 28]. While such techniques are highly effective, they demand deployment of special hardware, which limits their applicability.

Over the years, significant progress has been made in testing concurrent programs. CHESS [17] is a stateless bounded model checker that performs systematic stress testing to expose bugs in concurrent programs. It can be adopted to reproduce Heisenbugs. However, since CHESS was not designed for failure reproduction, it does not exploit available failure information to guide its enumeration of different schedules. Our technique leverages failure core dumps for this purpose. CTrigger [22] is another concurrency testing technique that searches for schedule perturbations to break usual patterns of shared variable accesses to expose faults. Random schedule perturbations are also shown to be effective in debugging races and deadlocks [27, 13]. We believe our core dump analysis can be synergistically combined with these algorithms.

9. Conclusion

We propose a concurrency bug reproduction technique for multi-core executions that relies on a novel core dump analysis and schedule search. The technique only requires adding loop counters to production runs, which has negligible runtime overhead. Given a failure core dump from a parallel (multi-core) run, our approach re-executes the program with the same input and identifies an execution point in the re-execution that corresponds to the failure point on a concurrent (single-core) system. This is done by reverse engineering a canonical state representation, called the execution index, of the failure point from the failure core dump. The index is used in the re-execution to locate the corresponding point. A new core dump is generated during the re-execution at the corresponding point. The two core dumps are compared to identify shared variables with different values, which imply schedule differences. A CHESS-like algorithm is proposed to leverage the shared variable difference information to search for failure inducing schedules. Experimental results show that the approach is very effective, produces failure inducing schedules more quickly than existing search techniques with modest overhead, and provides a feasible technique for reproducing bugs that manifest in multi-core environments.

References

[1] A. R. Alameldeen and D. A. Wood. Addressing Workload Variability in Architectural Simulations. In *IEEE Micro*, 23(6):94–98, 2003.

[2] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, pages 193-206, 2009.

[3] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *PLDI*, pages 201–212, 2005.

[4] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *VEE*, pages 154–163, 2006.

[5] H. J. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. In *Software Practice and Experience*, 18(9):807-820, 1988.

[6] M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *OOPSLA*, pages 97–112, 2007.

[7] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS*, pages 48–59, 1998.

[8] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE*, pages 121–130, 2008.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[10] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-preserving Data Publishing: A Survey on Recent Developments. In *ACM Computing Surveys*, 2009.

[11] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An Application-Level Kernel for Record and Replay. In *OSDI*, pages 193–208, 2008.

[12] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, pages 265–276, 2008.

[13] P. Joshi, C. S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *PLDI*, pages 110–120, 2009.

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *USENIX*, pages 1–15, 2005.

[15] B. Korel and J. Laski. Dynamic Program Slicing. In *Information Processing Letters*, 29(3):155–163, 1988.

[16] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, pages 73–84, 2009.

[17] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.

[18] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, pages 229–240, 2006.

[19] N. Nethercote and J. Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100, 2007.

[20] R. H. B. Netzer and M. H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *PLDI*, pages 313–325, 1994.

[21] D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[22] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *ASPLOS*, pages 25–36, 2009.

[23] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Do You Have to Reproduce the Bug at the First Replay Attempt? – pres: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, pages 177–192, 2009.

[24] M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. d. Kergommeaux, and D. Kranzlmüller. Record/Replay for Nondeterministic Program Executions. In *Communcation of the ACM*, 46(9):62–67, 2003.

[25] Y. Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *Automated Analysis-Driven Debugging*, pages 69–76, 2005.

[26] , S. Sarkar, P. Sewell, F.Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Aglave The Semantics of x86-CC Multiprocessor Machine Code In *POPL*, pages 379–391, 2009.

[27] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, pages 11–21, 2008.

[28] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension For Rollback and Deterministic Replay for Software Debugging. In *USENIX*, pages 29–44, 2004.

[29] B. Xin, N. Sumner, and X. Zhang. Efficient Program Execution Indexing. In *PLDI*, pages 238–249, 2008.

[30] X. Zhang, R. Gupta, and Y. Zhang. Cost and Precision Tradeoffs of Dynamic Data Slicing Algorithms. *ACM Transactions on Programming Languages and Systems*, 27(4):631–661, 2005.