Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures

Baris Kasikci[1] Benjamin Schubert[1] Cristiano Pereira[2] Gilles Pokam[2] George Candea[1]

[1]{baris.kasikci,benjamin.schubert,george.candea}@epfl.ch [2]{cristiano.l.pereira,gilles.a.pokam}@intel.com
[1] School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL)
[2] Intel Corporation

Abstract

Developers spend a lot of time searching for the root causes of software failures. For this, they traditionally try to reproduce those failures, but unfortunately many failures are so hard to reproduce in a test environment that developers spend days or weeks as ad-hoc detectives. The shortcomings of many solutions proposed for this problem prevent their use in practice.

We propose failure sketching, an automated debugging technique that provides developers with an explanation ("failure sketch") of the root cause of a failure that occurred in production. A failure sketch only contains program statements that lead to the failure, and it clearly shows the differences between failing and successful runs; these differences guide developers to the root cause. Our approach combines static program analysis with a cooperative and adaptive form of dynamic program analysis.

We built Gist, a prototype for failure sketching that relies on hardware watchpoints and a new hardware feature for extracting control flow traces (Intel Processor Trace). We show that Gist can build failure sketches with low overhead for failures in systems like Apache, SQLite, and Memcached.

1. Introduction

Debugging—the process of finding and fixing bugs—is time-consuming (around 50% [44] of the development time). This is because debugging requires a deep understanding of the code and the bug. Misunderstanding the code or the bug can lead to incorrect fixes, or worse, to fixes that introduce new bugs [39].

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10. . . $15.00.
http://dx.doi.org/10.1145/2815400.2815412

Traditionally, debugging is done in an iterative fashion: the developer runs the failing program over and over in a debugger, hoping to reproduce the failure, understand its root cause, and finally fix it. Fixing bugs generally requires the diagnosis of the root cause.

Intuitively, a root cause is the gist of the failure; it is a cause, or a combination of causes, which when removed from the program, prevents the failure associated with the root cause from recurring [74]. More precisely, a root cause of a failure is the negation of the predicate that needs to be enforced so that the execution is constrained to not encounter the failure [80].

The ability to reproduce failures is essential to traditional debugging, because developers rely on reproducing bugs to diagnose root causes. A recent study at Google [57] revealed that developers' ability to reproduce bugs is essential to fixing them. However, in practice, it is not always possible to reproduce bugs, and practitioners report that it takes weeks to fix hard-to-reproduce concurrency bugs [18].

The greatest challenge though, is posed by bugs that only recur in production and cannot be reproduced in-house. Diagnosing the root cause and fixing such bugs is truly hard. In [57], developers noted: *"We don't have tools for the once every 24 hours bugs in a 100 machine cluster."* An informal poll on Quora [54] asked *"What is a coder's worst nightmare,"* and the answers were *"The bug only occurs in production and can't be replicated locally,"* and *"The cause of the bug is unknown."*

A promising method to cope with hard to reproduce bugs is using record/replay systems [2, 46]. Record/replay systems record executions and allow developers to replay the failing ones. Even though these systems are helpful, they have not seen widespread adoption in production systems due to the high overheads of software-only systems [14, 68], or due to lack of the hardware support that they rely on [20, 46]. The recorded execution contains all statements present in the original run. While this is very useful, the recorded execution also contains statements that do not contribute to the failure. Root cause diagnosis may be more time consum-

ing and thereby more difficult if the recorded execution contains elements that do not pertain to the failure [42] (e.g., print statements that are not related to a failure). Ultimately record/replay systems merely aim to reproduce bugs, so root cause diagnosis remains a time-consuming developer task that is done manually.

Other approaches to bug reproduction and root cause diagnosis either assume knowledge of failing program inputs [50, 58, 76] or rely on special runtime support for checkpointing [67] or special hardware extensions [4, 52] that are not deployed. Some techniques sample the execution for root cause diagnosis [5, 25, 38] and therefore, may miss information present in the execution. This increases the latency of root cause diagnosis, especially for elusive bugs.

In this paper, we present failure sketching, a technique that automatically produces a high level execution trace called the *failure sketch* that includes the statements that lead to a failure and the differences between the properties of failing and successful program executions. Our evaluation shows that these differences, which are commonly accepted as pointing to root causes [38, 58, 81], indeed point to the root causes of the bugs we evaluated (§5).

Fig. 1 shows an example failure sketch for a bug in Pbzip2, a multithreaded compression tool [16]. Time flows downward, and execution steps are enumerated along the flow of time. The failure sketch shows the statements (in this case statements from two threads) that affect the failure and their order of execution with respect to the enumerated steps (i.e., the control flow). The arrow between the two statements in dotted rectangles indicates the difference between failing executions and successful ones. In particular, in failing executions, the statement `f->mut` from T_1 is executed before the statement `mutex_unlock(f->mut)` in T_2. In non-failing executions, `cons` returns before `main` sets `f->mut` to `NULL`. The failure sketch also shows the value of the variable `f->mut` (i.e., the data flow) in a dotted rectangle in step 7, indicating that this value is 0 in step 7 only in failing runs. A developer can use the failure sketch to fix the bug by introducing proper synchronization that eliminates the offending thread schedule. This is exactly how pbzip2 developers fixed this bug, albeit four months after it was reported [16].

The insight behind the work presented here is that failure sketches can be built automatically, by using a combination of static program analysis and cooperative dynamic analysis. The use of a brand new hardware feature in Intel CPUs helps keep runtime performance overhead low (3.74% in our evaluation).

We built a prototype, Gist, that automatically generates a failure sketch for a given failure. Given a failure, Gist first statically computes a program slice that contains program statements that can potentially affect the program statement where the failure manifests itself. Then, Gist performs data and control flow tracking in a cooperative setup by targeting

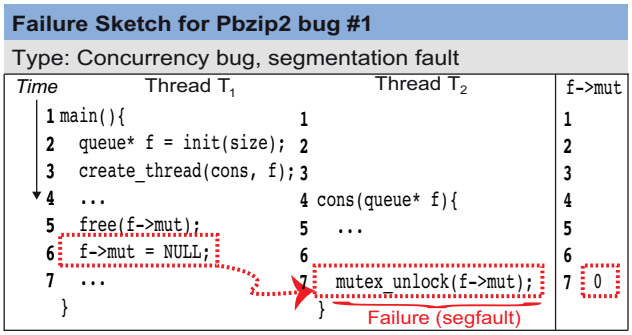

Figure 1: The failure sketch of pbzip2 bug.

either multiple executions of the same program in a data center or users who execute the same program. Gist uses hardware watchpoints to track the values of data items in the slice, and uses Intel Processor Trace [23] to trace the control flow.

This paper makes the following contributions:

- A ***low overhead technique to automatically build failure sketches*** by combining static analysis with cooperative and adaptive dynamic analysis.
- A ***first and practical demonstration*** of how Intel Processor Trace, a new technology that started shipping in early 2015 Broadwell processors [23], can be used to perform root cause diagnosis.

Although Gist relies on Intel Processor Trace and hardware watchpoints for practical and low-overhead control and data flow tracking, Gist's novelty is in the combination of static program analysis and dynamic runtime tracing to judiciously select how and when to trace program execution in order to best extract the information for failure sketches.

We evaluated our Gist prototype using 11 failures from 7 different programs including Apache, SQLite, and Memcached. The Gist prototype managed to automatically build failure sketches with an average accuracy of 96% for all the failures while incurring an average performance overhead of 3.74%. On average, Gist incurs 166× less runtime performance overhead than a state-of-the art record/replay system.

In the rest of the paper, we describe the challenges of root cause diagnosis (§2), Gist's design (§3) and our prototype implementation (§4), an evaluation on real-world applications (§5). We then discuss open questions and limitations (§6), related work (§7), and finally we conclude (§8).

2. Challenges

We now identify some of the key challenges of root cause diagnosis.

(1) ***Non-reproducible and infrequent bugs***: it is challenging to diagnose root causes of failures that are hard to reproduce in house. When developers can't reproduce fail-

ures, they have to "fill in the gaps" and potentially spend a lot of time.

This is exacerbated if the failure recurs rarely in production, but not rarely enough to be ignored or if it is a catastrophic bug [65]. This can slow down the process of gathering failure-related information from production runs (i.e., executions in a data center or at user endpoints), and ultimately delay root cause diagnosis. Existing root cause diagnosis techniques that rely on sampling program state during the execution increase the latency of root cause diagnosis.

(2) ***Accuracy***: it is not enough to quickly bootstrap the root cause diagnosis process, it is also necessary to eventually diagnose the root cause accurately. This is hard for complex software with many modules and interactions. Developers have limited time to do root cause diagnosis, therefore they should not be misled by inaccurate root cause diagnosis information.

Two main factors determine the accuracy of root cause diagnosis: the correct identification of relevant program statements that cause the failure and the correct identification of program properties (execution order of statements, data values, etc) that cause the failure.

(3) ***Overhead and intrusiveness***: any practical root cause diagnosis technique should incur low performance overhead in production and minimally perturb real-user executions.

To reduce overhead, existing root cause diagnosis techniques rely on special runtime and hardware support that is not readily available. Solutions that perturb the actual behavior of production runs nondeterministically may mask the bug frequently but not always, and thus make it harder to diagnose the root cause and remove the potential for (even occasional) failure [47].

Gist addresses the key challenges of root cause diagnosis using a hybrid static-dynamic analysis. To quickly bootstrap the root cause diagnosis process, Gist builds a first failure sketch after a single manifestation of a failure. This first failure sketch is not perfect: it lacks elements that may be useful for understanding the root cause of a bug, and it has some elements that are not needed for root cause diagnosis. Therefore, for bugs that recur, Gist gathers more control and data flow information from different production runs that encounter the same failure[1]. This step improves the accuracy of failure sketches by eliminating most of the aforementioned imperfections.

3. Design

Gist, our system for building failure sketches has three main components: the static analyzer, the failure sketch computation engine, and the runtime that tracks production runs. The static analyzer and the failure sketch computation engine constitutes the server side of Gist, and they can be centralized or distributed, as needed. The runtime constitutes the client-side of Gist, and it runs in a cooperative setting such as in a data center or in multiple users' machines, similar to RaceMob [32].

[1] Gist identifies the same failure across multiple executions by matching the program counters and stack traces of those executions.

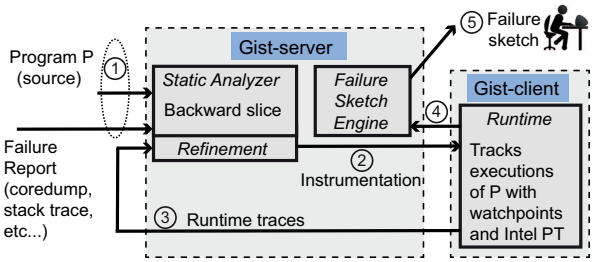

Figure 2: The architecture of Gist

The usage model of Gist is shown in Fig. 2. Gist takes as input a program (source code and binary) and a failure report ① (e.g., stack trace, the statement where the failure manifests itself, etc). Gist, being a developer tool has access to the source code. Using these inputs, Gist computes a backward slice [72] by computing the set of program statements that potentially affect the statement where the failure occurs. Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm. Then, Gist instructs its runtime, running in a data center or at user endpoints, ② to instrument the programs and gather more traces (e.g., branches taken and values computed at runtime). Gist then uses these traces to refine the slice ③: refinement removes from the slice the statements that don't get executed during the executions that Gist monitors, and it adds to the slice statements that were not identified as being part of the slice initially. Refinement also determines the inter-thread execution order of statements accessing shared variables and the values that the program computes. Refinement is done using hardware watchpoints for data flow and Intel Processor Trace (Intel PT) for control flow. Gist's failure sketch engine gathers execution information from failing and successful runs ④. Then, Gist determines the differences between failing and successful runs and builds a failure sketch ⑤ for the developer to use.

Gist operates in a cooperative setting [32, 38] where multiple instances of the same software execute in a data center or in multiple users' machines. Gist's server side (e.g., a master node) performs offline analysis and distributes instrumentation to its client side (e.g., a node in a data center). Gist incurs low overhead, so it can be kept always-on and does not need to resort to sampling an execution [38], thus avoiding missing information that can increase root cause diagnosis latency.

Gist operates iteratively: the instrumentation and refinement continues as long as developers see fit, continuously improving the accuracy of failure sketches. Gist generates a failure sketch after a first failure using static slicing. Our evaluation shows that, in some cases, this initial sketch is

sufficient for root cause diagnosis, whereas in other cases refinement is necessary (§3.2).

We now describe how each component of Gist works and explain how they solve the challenges presented in the previous section (§2). We first describe how Gist computes the static slice followed by slice refinement via adaptive tracking of control and data flow information. Then we describe how Gist identifies the root cause of a failure using multiple failing and successful runs.

3.1 Static Slice Computation

Gist uses an interprocedural, path-insensitive and flow-sensitive backward slicing algorithm to identify the program statements that may affect the statement where the failure manifests itself at runtime. We chose to make Gist's slicing algorithm interprocedural because failure sketches can span the boundaries of functions. The algorithm is path-insensitive in order to avoid the cost of path-sensitive analyses that do not scale well [3, 55]. However, this is not a shortcoming, because Gist can recover precise path information at runtime using low-cost control flow tracking (§3.2.2). Finally, Gist's slicing algorithm is flow-sensitive because it traverses statements in a specific order (backwards) from the failure location. Flow-sensitivity generates static slices with statements in the order they appear in the program text (except some out-of-order statements due to path-insensitivity, which are fixed using runtime tracking), thereby helping the developer to understand the flow of statements that lead to a failure.

Algorithm 1 describes Gist's static backward slicing: it takes as input a failure report (e.g., a coredump, a stack trace) and the program's source code, and it outputs a static backward slice. For clarity, we define several terms we use in the algorithm. CFG refers to the control flow graph of the program (Gist computes a whole-program CFG as we explain shortly). An item (line 7) is an arbitrary program element. A source (line 8, 16) is an item that is either a global variable, a function argument, a call, or a memory access. Items that are not sources are compiler intrinsics, debug information, and inline assembly. The definitions for a source and an item are specific to LLVM [36], which is what we use for the prototype (§4). The function `getItems` (line 1) returns all the items in a given statement (e.g., the operands of an arithmetic operation). The function `getRetValues` (line 11) performs an intraprocedural analysis to compute and return the set of items that can be returned from a given function call. The function `getArgValues` (line 14) computes and returns the set of arguments that can be used when calling a given function. The function `getReadOperand` (line 20) returns the item that is read, and the function `getWrittenOperand` (line 23) returns the item that is written.

Gist's static slicing algorithm differs from classic static slicing [72] in two key ways:

First, Gist addresses a challenge that arises for multi-threaded programs because of the implicit control flow edges

Algorithm 1: Backward slice computation (Simplified)

```
   Input: Failure report report, program source code program
   Output: Static backward slice slice
1  workSet ← getItems(failingStmt)
2  function init ()
3   failingStmt ← extractFailingStatement(report)
4  function computeBackwardSlice (failingStmt, program)
5   cfg ← extractCFG(program)
6   while !workSet.empty() do
7       item ← workSet.pop()
8       if isSource(item) then
9           slice.push(item)
10          if isCall(item) then
11              retValues ← getRetValues(item, cfg)
12              workSet ← workSet ∪ retValues
13          else if isArgument(item) then
14              argValues ← getArgValues(item, cfg)
15              workSet ← workSet ∪ argValues
16 function isSource (item)
17 if item is (global||argument||call||memory access) then
18      return true
19 else if item is read then
20      workSet ← workSet ∪ item.getReadOperand()
21      return true
22 else if item is write then
23      workSet ← workSet ∪ item.getWrittenOperand()
24      return true
25  return false
```

that get created due to thread creations and joins. For this, Gist uses a compiler pass to build the *thread interprocedural control flow graph* (TICFG) of the program [75]. An *interprocedural control flow graph* (ICFG) of a program connects each function's CFG with function call and return edges. TICFG then augments ICFG to contain edges that represent thread creation and join statements (e.g., a thread creation edge is akin to a callsite with the thread start routine as the target function). TICFG represents an overapproximation of all the possible dynamic control flow behaviors that a program can exhibit at runtime. TICFG is useful for Gist to track control flow that is implicitly created via thread creation and join operations (§3.2.2).

Second, unlike other slicing algorithms [59], Gist does not use static alias analysis. Alias analysis could determine an overapproximate set of program statements that may affect the computation of a given value and augment the slice with this information. Gist does not employ static alias analysis because, in practice, it can be over 50% inaccurate [32], which would increase the static slice size that Gist would have to monitor at runtime, thereby increasing its performance overhead. Gist compensates for the lack of alias anal-

ysis with runtime data flow tracking, which adds the statements that Gist misses to the static slice (§3.2.3).

The static slice that Gist computes has some extraneous items that do not pertain to the failure, because the slicing algorithm lacks actual execution information. Gist weeds out this information using accurate control flow tracking at runtime (§3.2.2).

3.2 Slice Refinement

Slice refinement removes the extraneous statements from the slice and adds to the slice the statements that could not be statically identified. Together with root cause identification (§3.3), the goal of slice refinement is to build what we call the ideal failure sketch.

> We define *an ideal failure sketch to be one that*: 1) contains only statements that have data and/or control dependencies to the statement where the failure occurs; 2) shows the failure predicting events that have the highest positive correlation with the occurrence of failures.

Different developers may have different standards as to what is the "necessary" information for root cause diagnosis; nevertheless, we believe that including all the statements that are related to a failure and identifying the failure predicting events constitute a reasonable and practical set of requirements for root cause diagnosis. Failure predictors are identified by determining the difference of key properties between failing and successful runs.

For example, failure sketches display the partial order of statements involved in data races and atomicity violations, however certain developers may want to know the total order of all the statements in an ideal failure sketch. In our experience, focusing on the partial order of statements that matter from the point of view of root cause diagnosis is more useful than having a total order of all statements. Moreover, obtaining the total order of all the statements in a failure sketch would be difficult without undue runtime performance overhead using today's technology.

We now describe Gist's slice refinement strategy in detail. We first describe adaptive tracking of a static slice to reduce the overhead of refinement (§3.2.1), then we describe how Gist tracks the control flow (§3.2.2) and the data flow (§3.2.3) to 1) add to the slice statements that get executed in production but are missing from the slice, and 2) remove from the slice statements that don't get executed in production.

3.2.1 Adaptive Slice Tracking

Gist employs Adaptive Slice-Tracking (AsT) to track increasingly larger portions of the static slice, until it builds a failure sketch that contains the root cause of the failure that it targets. Gist performs AsT by dynamically tracking control and data flow while the software runs in production. AsT does not track all the control and data elements in the static slice at once in order to avoid introducing performance overhead.

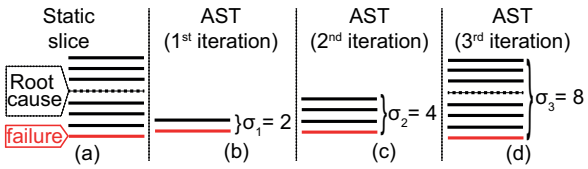

Figure 3: Adaptive slice tracking in Gist

It is challenging to pick the size of the slice, σ, to monitor at runtime, because 1) a too large σ would cause Gist to do excessive runtime tracking and increase overhead; 2) a too small σ may cause Gist to track too many runs before identifying the root cause, and so increase the latency of root cause diagnosis.

Based on previous observations that root causes of most bugs are close to the failure locations [53, 71, 82], Gist initially enables runtime tracking for a small number of statements ($\sigma = 2$ in our experiments) backward from the failure point. We use this heuristic because even a simple concurrency bug is likely to be caused by two statements from different threads. This also allows Gist to avoid excessive runtime tracking if the root cause is close to the failure (i.e., the common case). Nonetheless, to reduce the latency of root cause diagnosis, Gist employs a multiplicative increase strategy for further tracking the slice in other production runs. More specifically, Gist doubles σ for subsequent AsT iterations, until a developer decides that the failure sketch contains the root cause and instructs Gist to stop AsT.

Consider the static slice for a hypothetical program in Fig. 3.(a), which displays the failure point (bottom-most solid line) and the root cause (dashed line). In the first iteration (Fig. 3.(b)), AsT starts tracking $\sigma_1 = 2$ statements back from the failure location. Gist cannot build a failure sketch that contains the root cause of this failure by tracking 2 statements, as the root cause lies further backwards in the slice. Therefore, in the second and third iterations (Fig. 3.(c-d)), AsT tracks $\sigma_2 = 4$ and $\sigma_3 = 8$ statements, respectively. Gist can build a failure sketch by tracking 8 statements.

In summary, AsT is a heuristic to resolve the tension between performance overhead, root cause diagnosis latency, and failure sketch accuracy. We elaborate on this tension in our evaluation (§5). AsT does not limit Gist's ability to track larger slices and build failure sketches for bugs with greater root-cause-to-failure distances, although it may increase the latency of root cause diagnosis.

3.2.2 Tracking Control Flow In Hardware

Gist tracks control flow to increase the accuracy of failure sketches by identifying which statements from the slice get executed during the monitored production runs. Static slicing lacks real execution information such as dynamically computed call targets, therefore tracking the dynamic control flow is necessary for high accuracy failure sketches.

Static slicing and control flow tracking jointly improve the accuracy of failure sketches: control flow traces identify statements that get executed during production runs that Gist monitors, whereas static slicing identifies statements that have a control or data dependency to the failure. The intersection of these statements represents the statements that relate to the failure and that actually get executed in production runs. Gist statically determines the locations where control flow tracking should start and stop at runtime in order to identify which statements from the slice get executed.

Although control flow can be tracked in a relatively straightforward manner using software instrumentation [40], hardware facilities offer an opportunity for a design with lower overhead. Our design employs Intel PT, a set of new hardware monitoring features for debugging. In particular, Intel PT records the execution flow of a program and outputs a highly-compressed trace (~0.5 bits per retired assembly instruction) that describes the outcome of all branches executed by a program. Intel PT can be programmed to trace only user-level code and can be restricted to certain address spaces. Additionally, with the appropriate software support, Intel PT can be turned on and off by writing to processor-specific registers. Intel PT is currently available in Broadwell processors, and we control it using our custom kernel driver (§4). Future families of Intel processors are also expected to provide Intel PT functionality.

We explain how Gist tracks the statements that get executed via control flow tracking using the example shown in Fig. 4.(a). The example shows a static slice composed of three statements ($stmt_1$, $stmt_2$, $stmt_3$). The failure point is $stmt_3$. Let us assume that, as part of AsT, Gist tracks these three statements. At the high level, Gist identifies all entry points and exit points to each statement and starts and stops control-flow tracking at each entry point and at each exit point, respectively. Tracing is started to capture control flow if the statements in the static slice get executed, and is stopped once those statements complete execution. We use postdominator analysis to optimize out unnecessary tracking.

In this example, Gist starts its analysis with $stmt_1$. Gist converts the branch decision information to statement execution information using the technique shown in box I in Fig. 4.(a). It first determines bb_1, the basic block in which $stmt_1$ resides, and determines the predecessor basic blocks $p_{11}...p_{1n}$ of bb_1. The predecessor basic blocks of bb_1 are blocks from which control can flow to bb_1 via branches. As a result, Gist starts control flow tracking in each predecessor basic block $p_{11}...p_{1n}$ (i.e., entry points). If Gist's control flow tracking determines at runtime that any of the branches from these predecessor blocks to bb_1 was taken, Gist deduces that $stmt_1$ was executed.

Gist uses an optimization when a statement it already processed strictly dominates the next statement in the static slice. A statement d strictly dominates a statement n (written

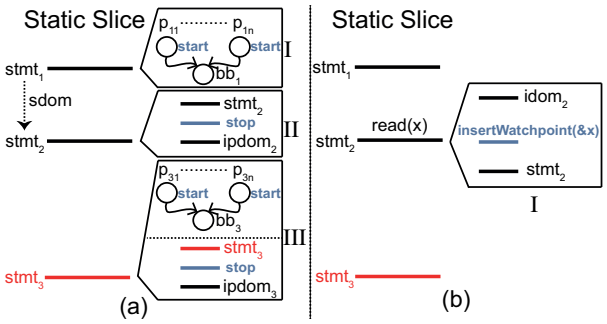

Figure 4: Example of control (a) and data (b) flow tracking in Gist. Solid horizontal lines are program statements, circles are basic blocks.

d *sdom* n) if every path from the entry node of the control flow graph to n goes through d, and d $\neq$ n. In our example, $stmt_1$ *sdom* $stmt_2$, therefore, Gist will have already started control flow tracking for $stmt_1$ when the execution reaches $stmt_2$, and so it won't need special handling to start control flow tracking for $stmt_2$.

However, if a statement that Gist processed does not strictly dominate the next statement in the slice, Gist stops control flow tracking. In our example, after executing $stmt_2$, since the execution may never reach $stmt_3$, Gist stops control flow tracking after $stmt_2$ gets executed. Otherwise tracking could continue indefinitely and impose unnecessary overhead. Intuitively, Gist stops control flow tracking *right after* $stmt_2$ gets executed as shown in box II of Fig. 4.(a). More precisely, Gist stops control flow tracking after $stmt_2$ and before $stmt_2$'s immediate postdominator. A node p is said to strictly postdominate a node n if all the paths from n to the exit node of the control flow graph pass through p, and n $\neq$ p. The immediate postdominator of a node n (*ipdom*(n)) is a unique node that strictly postdominates n and does not strictly postdominate any other strict postdominators of n.

Finally, as shown in box III in Fig. 4.(a), Gist processes $stmt_3$ using the combination of techniques it used for $stmt_1$ and $stmt_2$. Because control flow tracking was stopped after $stmt_2$, Gist first restarts it at each predecessor basic block $p_{31}...p_{3n}$ of the basic block bb_3 that contains $stmt_3$, then Gist stops it after the execution of $stmt_3$.

3.2.3 Tracking Data Flow in Hardware

Similar to control flow, data flow can also be tracked in software, however this can be prohibitively expensive [67]. Existing hardware support can be used for low overhead data flow tracking. In this section, we describe why and how Gist tracks data flow.

Determining the data flow in a program increases the accuracy of failure sketches in two ways:

First, Gist tracks the total order of memory accesses that it monitors to increase the accuracy of the control flow shown in the failure sketch. Tracking the total order is important

mainly for shared memory accesses from different threads, for which Intel PT does not provide order information. Gist uses this order information in failure sketches to help developers reason about concurrency bugs.

Second, while tracking data flow, Gist discovers statements that access the data items in the monitored portion of the slice that were missing from that portion of the slice. Such statements exist because Gist's static slicing does not use alias analysis (due to alias analysis' inaccuracy) for determining all statements that can access a given data item.

Gist uses hardware watchpoints present in modern processors to track the data flow (e.g., x86 has 4 hardware watchpoints [22]). They enable tracking the values written to and read from memory locations with low runtime overhead.

For a given memory access, Gist inserts a hardware watchpoint for the address of the accessed variable at a point *right before* the access instruction. More precisely, the inserted hardware watchpoint must be located before the access and after the immediate dominator of that access. Fig. 4.(b) shows an example, where Gist places a hardware watchpoint for the address of variable `x`, just before $stmt_2$ (`read(x)`).

Gist employs several optimizations to economically use its budget of limited hardware watchpoints when tracking the data flow. First, Gist only tracks accesses to shared variables, it does not place a hardware watchpoint for the variables allocated on the stack. Gist maintains a set of active hardware watchpoints to make sure to not place a second hardware watchpoint at an address that it is already watching.

If the statements in the slice portion that AsT monitors access more memory locations than the available hardware watchpoints on a user machine, Gist uses a cooperative approach to track the memory locations across multiple production runs. In a nutshell, Gist's collaborative approach instructs different production runs to monitor different sets of memory locations in order to monitor all the memory locations that are in the slice portion that Gist monitors. However, in practice, we did not encounter this situation (§5).

3.3 Identifying the Root Cause

In this section, we describe how Gist determines the differences of key execution properties (i.e., control and data flow) between failing and successful executions in order to do root cause diagnosis.

For root cause diagnosis, Gist follows a similar approach to cooperative bug isolation [4, 25, 38], which uses statistical methods to correlate failure predictors to failures in programs. A failure predictor is a predicate that, when true, predicts that a failure will occur [37]. Carefully crafted failure predictors point to failure root causes [38].

Gist-generated failure sketches contain a set of failure predictors that are both informative and good indicators of failures. A failure predictor is informative if it contains

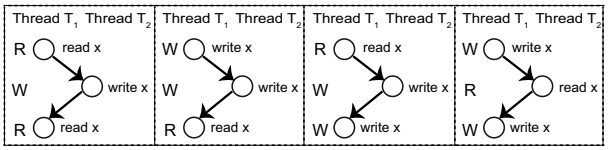

Figure 5: Four common atomicity violation patterns (RWR, WWR, RWW, WRW). Adapted from [4].

enough information regarding the failure (e.g., thread schedules, critical data values). A failure predictor is a good indicator of a failure if it has high positive correlation with the occurrence of the failure.

Gist defines failure predictors for both sequential and multithreaded programs. For sequential programs, Gist uses branches taken and data values computed as failure predictors. For multithreaded programs, Gist uses the same predicates it uses for sequential programs, as well as special combinations of memory accesses that portend concurrency failures. In particular, Gist considers the common single-variable atomicity violation patterns shown in Fig. 5 (i.e., RWR (Read, Write, Read), WWR, RWW, WRW) and data race patterns (WW, WR, RW) as concurrency failure predictors.

For both failing and successful runs, Gist logs the order of accesses and the value updates to shared variables that are part of the slice it tracks at runtime. Then, using an offline analysis, Gist searches the aforementioned failure-predicting memory access patterns in these access logs. Gist associates each match with either a successful run or a failing run. Gist is not a bug detection tool, but it can understand common failures, such as crashes, assertion violations, and hangs. Other types of failures can either be manually given to Gist, or Gist can be used in conjunction with a bug finding tool.

Once Gist has gathered failure predictors from failing and successful runs, it uses a statistical analysis to determine the correlation of these predictors with the failures. Gist computes the precision P (how many runs fail among those that are predicted to fail by the predictor?), and the recall R (how many runs are predicted to fail by the predictor among those that fail?). Gist then ranks all the events by their F-measure, which is the weighted harmonic mean of their precision and recall $F_\beta = (1+\beta^2)\frac{P.R}{\beta^2.P+R}$ to determine the best failure predictor. Gist favors precision by setting β to 0.5 (a common strategy in information retrieval [56]), because its primary aim is to not confuse the developers with potentially erroneous failure predictors (i.e., false positives).

The failure sketch presents the developer with the highest-ranked failure predictors for each type of failure predictor (i.e., branches, data values, and statement orders). An example of this is shown in Fig. 1, where the dotted rectangles show the highest-ranked failure predictor.

As an example, consider the execution trace shown in Fig. 6.(a). Thread T_1 reads `x`, after which thread T_2 gets

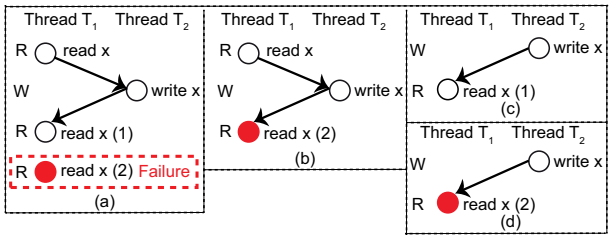

Figure 6: A sample execution failing at the second read in T_1 (a), and three potential concurrency errors: a RWR atomicity violation (b), 2 WR data races (c-d).

scheduled and writes to `x`. Then T_1 gets scheduled back and reads `x` twice in a row, and the program fails (e.g., the second read could be made as part of an assertion that causes the failure). This execution trace has three memory access patterns that can potentially be involved in a concurrency bug: a RWR atomicity violation in Fig. 6.(b) and two data races (or order violations) in Fig. 6.(c) and 6.(d). For this execution, Gist logs these patterns and their outcome (i.e., failure and success: 6.(b) and 6.(d) fail, whereas the pattern in 6.c succeeds. Gist keeps track of the outcome of future access patterns and computes their F-measure to identify the highest ranked failure predictors.

There are two key differences between Gist and cooperative bug isolation (CBI). First, Gist tracks all data values that are part of the slice that it monitors at runtime, allowing it to diagnose the root cause of failures caused by a certain input, as opposed to CBI, which tracks ranges of some variables. Second, Gist uses different failure predictors than CCI [25] and PBI [4], which allow developers to distinguish between different kinds of concurrency bugs, whereas PBI and CCI use the same predictors for failures with different root causes (e.g., invalid MESI ([49] state for all of RWR, WWR, RWW atomicity violations)).

4. Implementation

In this section, we describe several implementation details of our Gist prototype.

We implemented Gist in 2,673 lines of C++ code for static analyses and instrumentation, 664 lines of C code for the Intel PT kernel driver [34], and 3,165 lines of Python code for the cooperative framework. We also built a 10,518-lines C++ simulator for Intel PT based on PIN [40], and used this simulator to evaluate the overhead of control flow tracking in software.

Gist's static slicing algorithm is built on the LLVM framework [36]. As part of this algorithm, Gist first augments the intraprocedural control flow graphs of each function with function call and return edges to build the interprocedural control flow graph (ICFG) of the program. Then, Gist processes thread creation and join functions (e.g., `pthread_create`, `pthread_join`) to determine which start routines the thread creation functions may call at runtime and where those routines will join back to their callers, using data structure analysis [35]. Gist augments the edges in the ICFGs of the programs using this information about thread creation/join in order to build the thread interprocedural control flow graph (TICFG) of the program. Gist uses the LLVM information flow tracker [27] as a starting point for its slicing algorithm.

Gist currently inserts a small amount of instrumentation into the programs it runs, mainly to start/stop Intel PT tracking and place a hardware watchpoint. To distribute the instrumentation, Gist uses `bsdiff` to create a binary patch file that it ships off to user endpoints or to a data center. We plan to investigate more advanced live update systems such as POLUS [11]. Another alternative is to use binary rewriting frameworks such as DynamoRio [9] or Paradyn [45].

Trace collection is implemented via a Linux kernel module which we refer to as the Intel PT kernel driver. The kernel driver configures and controls the hardware using the documented MSR (Machine Specific Register) interface. The driver allows filtering of what code is traced using the privilege level (i.e. kernel vs. user-space) and CR3 values, thus allowing tracing of individual processes. The driver uses a memory buffer sized at 2 MB, which is sufficient to hold traces for all the applications we have tested. Finally, Gist-instrumented programs use an `ioctl` interface that our driver provides to turn tracing on/off.

Gist's hardware watchpoint use is based on the `ptrace` system call. Once Gist sets the desired hardware watchpoints, it detaches from the program (using the PTRACE_-DETACH), thereby not incurring any performance overhead. Gist's instrumentation handles hardware watchpoint triggers atomically in order to maintain a total order of accesses among memory operations. Gist logs the program counter when a hardware watchpoint is hit, which it later translates into source line information at developer site. Gist does not need debug information to do this mapping: it uses the program counter and the offset at which the program binary is loaded to compute where in the actual program this address corresponds to.

5. Evaluation

In this section we aim to answer the following questions about Gist and failure sketching: Is Gist capable of automatically computing failure sketches (§5.1)? Are these sketches accurate (§5.2)? How efficient is the computation of failure sketches in Gist (§5.3)?

To answer these questions we benchmark Gist with several real world programs: Apache httpd [21] is a popular web server. Cppcheck [43] is a C/C++ static analysis tool integrated with popular development tools such as Visual Studio, Eclipse, and Jenkins. Curl [62] is a data transfer tool for network protocols such as FTP and HTTP, and it is part of most Linux distributions and many programs, like LibreOffice and CMake. Transmission [66] is the default BitTor-

Bug name / software	Software version	Software size [LOC]	Bug ID from bug DB	Static slice size, in source LOC (LLVM instructions)	Ideal failure sketch size, in source LOC (LLVM instrs)	Gist-computed sketch size, in source LOC (LLVM instrs)	Duration of failure sketch computation by Gist: # failure recurrences <time> (offline analysis time)
Apache-1	2.2.9	224,533	45605	7 (23)	8 (23)	8 (23)	5 <4m:22s> (1m:28s)
Apache-2	2.0.48	169,747	25520	35 (137)	4 (16)	4 (16)	4 <3m:53s> (0m:55s)
Apache-3	2.0.48	169,747	21287	354 (968)	6 (6)	8 (8)	3 <4m:17s> (1m:19s)
Apache-4	2.0.46	168,574	21285	335 (805)	9 (12)	13 (16)	4 <5m:34s> (1m:23s)
Cppcheck-1	1.52	86,215	3238	3,662 (10,640)	11 (16)	11 (16)	4 <5m:14s> (2m:32s)
Cppcheck-2	1.48	76,009	2782	3,028 (8,831)	3 (8)	3 (8)	3 <3m:21s> (1m:40s)
Curl	7.21	81,658	965	15 (46)	6 (17)	6 (17)	5 <1m:31s> (0m:40s)
Transmission	1.42	59,977	1818	680 (1,681)	2 (7)	3 (8)	3 <0m:23s> (0m:17s)
SQLite	3.3.3	47,150	1672	389 (1,011)	3 (4)	3 (4)	2 <2m:47s> (1m:43s)
Memcached	1.4.4	8,182	127	237 (1,003)	6 (13)	8 (16)	4 <0m:56s> (0m:02s)
Pbzip2	0.9.4	1,492	N/A	8 (14)	6 (13)	9 (14)	4 <1m:12s> (0m:03s)

Table 1: Bugs used to evaluate Gist. Bug IDs come from the corresponding official bug database. LOC are measured using sloccount [73]. We report slice and sketch sizes in both source code lines and LLVM instructions. Time is reported in minutes:seconds.

rent client in Ubuntu and Fedora Linux, as well as Solaris. SQLite [60] is an embedded database used in Chrome, Firefox, iOS, and Android. Memcached is a distributed memory object cache system used by services such as Facebook and Twitter [15]. Pbzip2 [16] is a parallel file compression tool.

We developed an extensible framework called Bugbase [7] in order to reproduce the known bugs in the aforementioned software. Bugbase can also be used to do performance benchmarking of various bug finding tools. We used Bugbase to obtain our experimental results.

We benchmark Gist on bugs (from the corresponding bug repositories) that were used by other researchers to evaluate their bug finding and failure diagnosis tools [5, 31, 50]. Apart from bugs in Cppcheck and Curl, all bugs are concurrency bugs. We use a mixture of workloads from actual program runs, test suites, test cases devised by us and other researchers [77], Apache's benchmarking tool `ab`, and SQLite's test harness. We gathered execution information from a total of 11,360 executions.

The distributed cooperative setting of our test environment is simulated, as opposed to employing real users, because CPUs with Intel PT support are still scarce, having become available only recently. In the future we plan to use a real-world deployment. Altogether we gathered execution information from 1,136 simulated user endpoints. Client-side experiments were run on a 2.4 GHz 4 core Intel i7-5500U (Broadwell) machine running a Linux kernel with an Intel PT driver [24]. The server side of Gist ran on a 2.9 GHz 32-core Intel Xeon E5-2690 machine with 256 GB of RAM running Linux kernel 3.13.0-44.

5.1 Automated Generation of Sketches

For all the failures shown in Table 1, Gist successfully computed the corresponding failure sketches after gathering execution information from 11,360 runs in roughly 35 minutes. The results are shown in the rightmost two columns. We verified that, for all sketches computed by Gist, the failure predictors with the highest F-measure indeed correspond to the root causes that developers chose to fix.

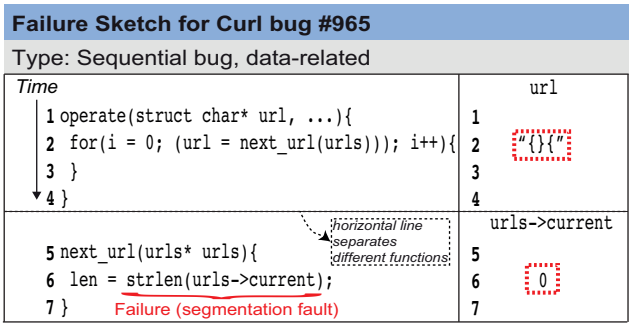

Figure 7: The failure sketch of Curl bug #965.

In the rest of this section, we present two failure sketches computed by Gist, to illustrate how developers can use them for root cause diagnosis and for fixing bugs. These two complement the failure sketch for the Pbzip2 bug already described in §1. Aside from some formatting, the sketches shown in this section are exactly the output of Gist. We renamed some variables and functions to save space in the figures. The statements or variable values in dotted rectangles denote failure predicting events with the highest F-measure values. We integrated Gist with KCachegrind [70], a call graph viewer that allows easy navigation of the statements in the failure sketch.

Fig. 7 shows the failure sketch for Curl bug #965, a sequential bug caused by a specific program input: passing the string "{}{" (or any other string with unbalanced curly braces) to Curl causes the variable `urls->current` in function `next_url` to be `NULL` in step 6. The value of `url` in step 2 ("{}{") and the value of `urls->current` in step 6 (0) are the best failure predictors. This failure sketch suggests that fixing the bug consists of either disallowing un-

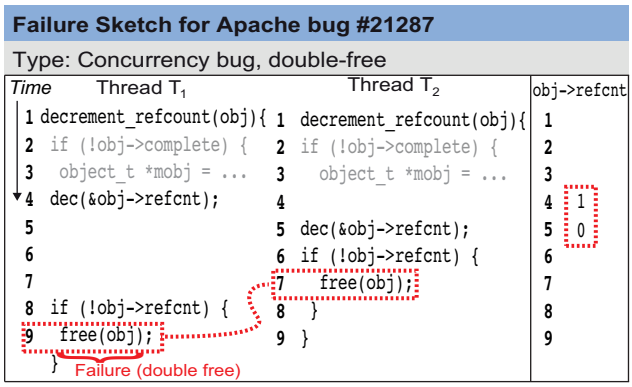

Figure 8: The failure sketch of Apache bug #21287. The grayed-out components are not part of the ideal failure sketch, but they appear in the sketch that Gist automatically computes.

balanced parentheses in the input url, or not calling `strlen` when `urls->current` is `NULL`. Developers chose the former solution to fix this bug [61].

Fig. 8 shows the failure sketch for Apache bug 21287, a concurrency bug causing a double free. The failure sketch shows two threads executing the `decrement_refcount` function with the same `obj` value. The `dec` function decrements `obj->refcount`. The call to `dec`, the `if` condition checking, namely `!obj->refcount`, and the call to `free` are not atomic, and this can cause a double free if `obj->refcount` is 0 in step 6 in T_2 and step 8 in T_1. The values of `obj->refcount` in steps 4 and 5 (1 and 0 respectively), and the double call to `free(obj)` are the best failure predictors. Developers fixed this bug by ensuring that the decrement-check-free triplet is executed atomically [63].

The grayed-out statements in the failure sketch in Fig. 8 are not part of the ideal failure sketch. The adaptive slice tracking part of Gist tracks them during slice refinement, because Gist does not know the statements in the ideal failure sketch a priori. For the Curl bug in Fig. 7, we do not show any grayed-out statements, because, adaptive slice tracking happens to track only the statements that are in the ideal failure sketch.

5.2 Failure Sketch Accuracy

In this section, we measure the accuracy (A) of failure sketches computed by Gist (Φ_G), as compared to ideal failure sketches that we computed by hand (Φ_I), according to our ideal failure sketch definition (§3.2). We define two components of failure sketch accuracy:

1) ***Relevance*** measures the extent to which a failure sketch contains all the statements from the ideal sketch and no other statements. We define relevance as the ratio of the number of LLVM instructions in $\Phi_G \cap \Phi_I$ to the number of statements in $\Phi_G \cup \Phi_I$. We compute relevance accuracy as a percentage, and define it as $A_R = 100 \cdot \frac{|\Phi_G \cap \Phi_I|}{|\Phi_G \cup \Phi_I|}$

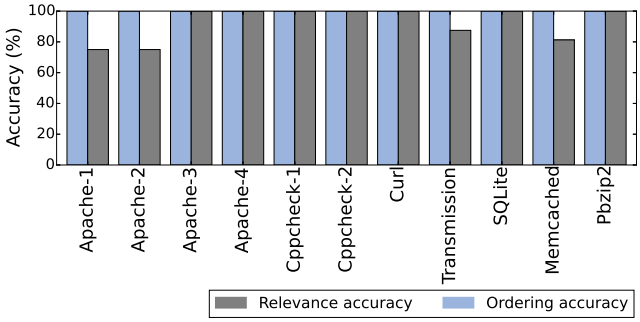

Figure 9: Accuracy of Gist, broken down into relevance accuracy and ordering accuracy.

2) ***Ordering*** measures the extent to which a failure sketch correctly represents the partial order of LLVM memory access instructions in the ideal sketch. To measure the similarity in ordering between the Gist-computed failure sketches and their ideal counterparts, we use the normalized Kendall tau distance [33] τ, which measures the number of pairwise disagreements between two ordered lists. For example, for ordered lists <A, B, C> and <A, C, B>, the pairs (A, B) and (A, C) have the same ordering, whereas the pair (B, C) has different orderings in the two lists, hence $\tau = 1$. We compute the ordering accuracy as a percentage defined by $A_O = 100 \cdot (1 - \frac{\tau(\Phi_G, \Phi_I)}{\text{\# of pairs in } \Phi_G \cap \Phi_I})$. Note that # of pairs in $\Phi_G \cap \Phi_I$ can't be zero, because both failure sketches will at least contain the failing instruction as a common instruction.

We define overall accuracy as $A = \frac{A_R + A_O}{2}$, which equally favors A_O and A_R. Of course, different developers may have different subjective opinions on which one matters most.

We show Gist's accuracy results in Fig. 9. Average relevance accuracy is 92%, average ordering accuracy is 100%, and average overall accuracy is 96%, which leads us to conclude that Gist can compute failure sketches with high accuracy. The accuracy results are deterministic from one run to the next.

Note that, for all cases when relevance accuracy is below 100%, it is because Gist's failure sketches have (relative to the ideal sketches) some excess statements in the form of a prefix to the ideal failure sketch, as shown in gray in Fig. 8. We believe that developers find it significantly easier to visually discard excess statements clustered as a prefix than excess statements that are sprinkled throughout the failure sketch, so this inaccuracy is actually not of great consequence.

We show in Fig. 10 the contribution of Gist's three analysis and tracking techniques to overall sketch accuracy. To obtain these measurements, we first measured accuracy when using just static slicing, then enabled control flow tracking and re-measured, and finally enabled also data flow tracking and re-measured. While the accuracy results are consistent across runs, the individual contributions may vary if, for ex-

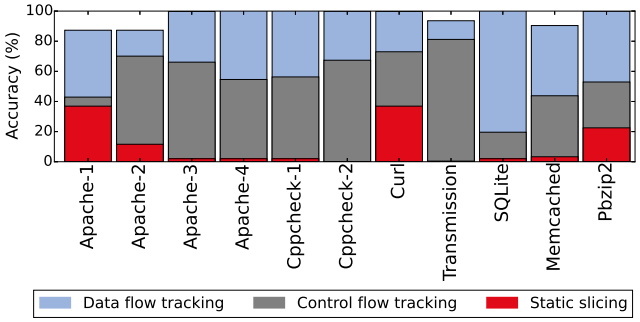

Figure 10: Contribution of various techniques to Gist's accuracy.

ample, workload non-determinism causes different paths to be exercised through the program.

A small contribution of a particular technique does not necessarily mean that it does not perform well for a given program, but it means that the other techniques that Gist had enabled prior to this technique "stole its thunder" by being sufficient to provide high accuracy. For example, in the case of Apache-1, static analysis performs well enough that control flow tracking does not need to further refine the slice. However, in some cases (e.g., for SQLite), tracking the inter-thread execution order of statements that access shared variables using hardware watchpoints is crucial for achieving high accuracy.

We observe that the amount of individual contribution varies substantially from one program to the next, which means that neither of these techniques would achieve high accuracy for all programs on its own, and so they are all necessary if we want high accuracy across a broad spectrum of software.

5.3 Efficiency

Now we turn our attention to the efficiency of Gist: how long does it take to compute a failure sketch, how much runtime performance overhead does it impose on clients, and how long does it take to perform its offline static analysis. We also look at how these measures vary with different parameters.

The last column of Table 1 shows Gist's failure sketch computation latency broken down into three components. We show the number of failure recurrences required to reach the best sketch that Gist can compute, and this number varies from 2 to 5 recurrences. We then show the total time it took in our simulated environment to find this sketch; this time is always less than 6 minutes, varying from <0m:23s> to <5m:34s>. Not surprisingly, this time is dominated by how long it takes the target failure to recur, and in practice this depends on the number of deployed clients and the variability of execution circumstances. Nevertheless, we present the values for our simulated setup to give an idea as to how long it took to build a failure sketch for each bug in our evaluation. Finally, in parentheses we show Gist's offline analysis time,

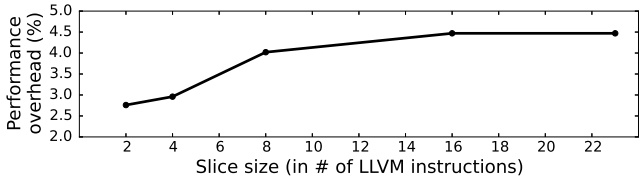

Figure 11: Gist's average runtime performance overhead across all runs as a function of tracked slice size.

which consists of computing the static slice plus generating instrumentation patches. This time is always less than 3 minutes, varying between <0m:2s> and <2m:32s>. We therefore conclude that, compared to the debugging latencies experienced by developers today, Gist's automated approach to root cause diagnosis presents a significant advantage.

In the context of adaptive slice tracking, the overhead incurred on the client side increases monotonically with the size of the tracked slice, which is not surprising. Fig. 11 confirms this experimentally. The portion of the overhead curve between the slice sizes 16 and 22 is relatively flat compared to the rest of the curve. This is because, within that interval, Gist only tracks a few control flow events for Apache-1 and Curl (these programs have no additional data flow elements in that interval), which introduces negligible overhead.

The majority of the overhead incurred on the client side stems from control flow tracking. In particular, the overhead of control flow tracking varies from a low of 2.01% to a high of 3.43%, whereas the overhead of data flow tracking varies from a low of 0.87% to a high of 1.04%.

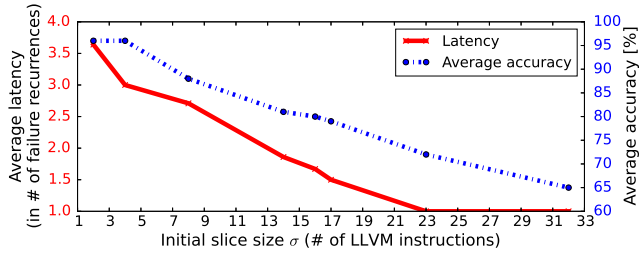

Figure 12: Tradeoff between slice size and the resulting accuracy and latency. Accuracy is in percentage, latency is in the number of failure recurrences.

What is perhaps not immediately obvious is the tradeoff between initial slice size σ and the resulting accuracy and latency. In Fig. 12, we show the average failure sketch accuracy across all programs we measured (right y-axis) and Gist's latency in # of recurrences (left y-axis) as a function of σ that Gist starts with (x-axis). As long as the initial slice size is less than the one for the best sketch that Gist can find, Gist's adaptive approach is capable of guiding the developer to the highest accuracy sketch. Of course, the time it takes to find the sketch is longer the smaller the starting

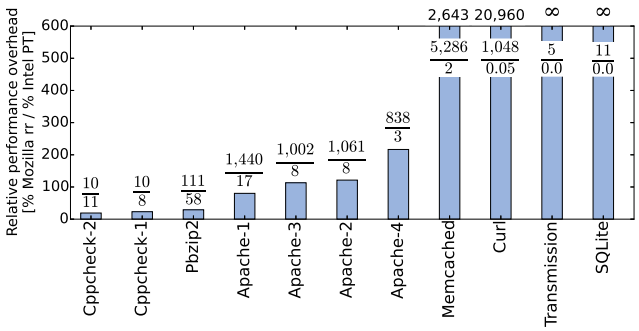

Figure 13: Comparison of the full tracing overheads of Mozilla rr and Intel PT.

slice size is, because the necessary # of recurrences is higher. There is thus an incentive to start with a larger slice size. Unfortunately, if this size overshoots the size of the highest accuracy sketch, then the accuracy of the outcome suffers, because the larger slice includes extraneous elements.

As we mentioned in §5.2, the extraneous statements that can lower Gist's accuracy are clustered as a prefix to the ideal failure sketch, allowing developers to easily ignore them. Therefore, if lower root cause diagnosis latency is paramount to the developers, they are comfortable ignoring the prefix of extraneous statements, and they can tolerate the slight increase in Gist's overhead, it is reasonable to configure Gist to start with a large σ (e.g., $\sigma = 23$ achieves a latency of *one failure recurrence* for all our benchmarks).

For the benchmarks in our evaluation, starting AsT at $\sigma = 4$ would achieve the highest average accuracy at the lowest average latency of 3, with an average overhead of 3.98%.

Finally, Fig. 13 compares Intel PT, the hardware-based control flow tracking mechanism we use in Gist, to Mozilla rr, a software-based state-of-the-art record & replay system. In particular, we compare the performance overhead imposed by the two tracking mechanisms on the client application. The two extremes are Cppcheck, where Mozilla rr is on par with Intel PT, and Transmission and SQLite, where Mozilla rr's overhead is over many orders of magnitude higher than Intel PT's [2]. For the benchmarks in our evaluation, full tracing using Intel PT incurs an average overhead of 11%, whereas full program record & replay incurs an average runtime overhead of 984%. Unlike Intel PT, Mozilla rr also gathers data flow information, but with Gist we have shown that full program tracing is not necessary for automating root cause diagnosis.

In conclusion, our empirical evaluation shows that Gist, a failure sketching prototype, is capable of automatically computing failure sketches for failures caused by real bugs in real systems (§5.1), these sketches have a high accuracy of 96% on average (§5.2), and the average performance overhead of failure sketching is 3.74% with $\sigma = 2$ (§5.3). We therefore believe failure sketching to be a promising approach for helping developers debug elusive bugs that occur only in production.

[2] Full tracing overheads of Transmission and SQLite are too low to be reliably measured for Intel PT, thus they are shown as 0%, and the corresponding Mozilla rr/Intel PT overheads for these systems are shown as ∞.

6. Discussion and Limitations

In this section, we discuss Gist's limitations, some remaining open questions and future work.

Intel PT is a mechanism we used in our failure sketching prototype Gist. It is instrumental for achieving low overhead control flow tracking, and therefore is important for building a practical tool. However, failure sketching is completely independent from Intel PT; it can be entirely implemented using software instrumentation, although our experiments using our Intel PT software simulator yielded runtime performance overheads that range from 3× to 5,000×. Furthermore, failure sketching's novelty is in combining heavy static program analysis with lightweight runtime monitoring to help developers perform root cause diagnosis.

Intel PT has certain shortcomings that Gist compensates using a combination of other mechanisms. First, Intel PT traces are partially ordered per CPU core, whereas diagnosing the root cause of most concurrency bugs requires having a control flow trace that is totally ordered across the CPU cores. Second, Intel PT only traces control flow, and does not contain any data values. Gist addresses these first two challenges using hardware watchpoints. The third challenge is that Intel PT may trace statements that do not necessarily pertain to the failure, which Gist mitigates by using static analysis. Finally, Intel PT's tracing overhead can be prohibitive if it is left always on, which Gist manages using a combination of static analysis and adaptive slice tracking.

Full control flow tracking using Intel PT will have even lower runtime overheads in future generations of Intel processors [8] after Broadwell. Therefore, in the near future, it is conceivable to have Intel PT tracing always on for some applications. Regardless, Gist can be used to further reduce the overhead to unnoticeable levels, and reducing the amount of collected information is especially useful for highly concurrent software, where the trace volume will be larger. Furthermore, the combination of static analysis and adaptive slice tracking is necessary to utilize the scarce number of hardware watchpoints judiciously. Regardless of performance improvements, Gist will remain useful for producing concise failure sketches.

We argue that additional hardware support would improve Gist's performance even further. For example, if Intel Processor Trace also captured a trace of the data addresses and values along with the control-flow, we could eliminate the need for hardware watchpoints and the complexity of a cooperative approach.

Unlike some other root cause diagnosis approaches [37, 58], Gist does not track predicates on data values such as ranges and inequalities, but it simply tracks data values themselves. As future work, we plan to track range and inequality predicates in Gist to provide richer information on data values.

A cooperative framework like Gist can have privacy implications depending on its exact deployment setup. If Gist targets a data center setting, there are fewer privacy concerns, as generally all the data that programs operate on is already within the data center. We plan to investigate ways to quantify and anonymize [64] the amount of information Gist ships from production runs at user endpoints to Gist's server.

Using ptrace for placing hardware watchpoints has challenges and limitations with respect to performance and usability. With regards to performance, using ptrace to place a hardware watchpoint incurs the overhead of the `ptrace` system call. In the future, this overhead can be mitigated using a user space instruction (similar to RDTSC [22]). With regards to usability, if a program is already using ptrace, Gist cannot attach to it using ptrace to place the hardware watchpoints necessary for data flow tracking. This situation can be remedied using several strategies. One option is to augment the existing ptrace functionality in the program to support the placement of hardware watchpoints. Another strategy is to use a third party interface (e.g., a new syscall or an `ioctl`) to place the hardware watchpoints.

Gist does not track variables that are allocated on the stack. It is technically possible for a thread to allocate memory on the stack and communicate the address of this memory to other threads, which can then use this address to share data. However, sharing stack addresses can make small programming errors catastrophic and difficult to isolate [10].

Debugging using failure sketches is strictly complementary to existing debugging techniques. In particular, we argue that failure sketches always help the debugging effort of developers. Failure sketches can also augment existing bug fixing techniques. For example, developers can use failure sketches to help tools like CFix [26] automatically synthesize fixes for concurrency bugs.

Finally, as future work, we plan to evaluate the effectiveness and efficiency of Gist in a real-world scenario, either in a data center or at user endpoints.

7. Related Work

In this section, we review a variety of techniques that have been developed to date to understand the root causes of failures and to help developers with debugging.

Delta debugging [81] isolates program inputs and variable values that cause a failure by systematically narrowing the state difference between a failing and a successful run. Delta debugging achieves this by repeatedly reproducing the failing and successful run, and altering variable values. Delta debugging has also been extended to isolate failure-inducing control flow information [13]. As opposed to delta debugging, Gist targets bugs that are hard to reproduce and aims to build a (potentially imperfect) sketch even with a single failing execution. If failures recur at the user endpoint, Gist can build more accurate sketches.

Gist's cooperative approach is inspired by work on cooperative bug isolation techniques such as CBI [38], CCI [25], PBI [4], LBRA/LCRA [5]. Gist builds upon the statistical techniques introduced in this line of work. Gist uses different failure predicting events for multithreaded code than these systems, to allow developers to differentiate between different types of concurrency bugs. Gist has low overhead, so it is always on and has low root cause diagnosis latency, because it does not rely on sampling like CBI, CCI, PBI. Unlike LCRA, Gist does not rely on a custom hardware extension, but it uses Intel PT, a hardware feature present in commodity processors. LBRA/LCRA works well for bugs with short root cause to failure distances, whereas failure sketch sizes are only limited by persistent storage size. Gist uses a different F-measure for failure ranking that favors precision over recall, because it aims to avoid false positives in root cause diagnosis. LBRA/LCRA preserves the privacy of users to some extent, because it does not track the data flow of a program, whereas Gist does not have mechanisms to protect privacy (although in theory it could), therefore making Gist more suitable to use cases where privacy is not a major concern.

Gist is also inspired by Windows Error Reporting (WER) [17], a large-scale cooperative error reporting system operating at Microsoft. After a failure, WER collects snapshots of memory and processes them using a number of heuristics (e.g., classification based on call stacks and error codes) to cluster reports that likely point to the same bug. WER can use failure sketches built by Gist to improve its clustering of bugs, and help developers fix the bugs faster.

Symbiosis [41] uses a technique called differential schedule projections that displays the set of data flows and memory operations that are responsible for a failure in a multithreaded program. Symbiosis profiles a failing program's schedule and generates non-failing alternate schedules. Symbiosis then determines the data flow differences between the failing schedule and the non-failing schedule in order to help developers identify root causes of failures. Unlike Symbiosis, Gist does not assume that it has access to a failing program execution that can be reproduced in-house. Furthermore, the statistical analysis in Gist allows root cause diagnosis of sequential bugs, whereas Symbiosis is targeted towards concurrency bugs only.

Previous work explored adaptive monitoring for testing and debugging. SWAT [19] adaptively samples program segments at a rate that is inversely proportional to their execution frequency. RaceTrack [78] adaptively monitors parts of a program that are more likely to harbor data races. Bias

free sampling [29] allows a developer to provide an adaptive scheme for monitoring a program's behavior. Adaptive bug isolation [6] uses heuristics to adaptively estimate and track program behaviors that are likely predictors of failures. Gist relies on static analysis to bootstrap and guide its adaptive slice monitoring, thereby achieving low latency and low overhead in root cause diagnosis.

PRES [50] records execution sketches, which are abstractions of real executions (e.g., just an execution log of functions), and performs state space exploration on those sketches to reproduce failures. Failure sketches also abstract executions. Unlike PRES, Gist helps developers do root diagnosis.

HOLMES [12] uses path profiles to perform bug isolation. HOLMES does not track any data values, whereas Gist relies on tracking data values for performing root cause diagnosis of concurrency bugs involving shared variables and also for providing richer debugging information to developers.

SherLog [79] uses a combination of program analysis and execution logs from a failed production run in order to automatically generate control and data flow information that aims to help developers diagnose the root causes of errors. Unlike Gist, Sherlog relies on logging to be always enabled at execution time, and works only for single-threaded programs.

ConSeq [82] computes static slices to identify shared memory reads starting from potentially failing statements (e.g., `assert`). It then records correct runs and, during replay, it perturbs schedules around shared memory reads to try to uncover bugs. Gist uses static slicing to identify all control and data dependencies to the failure point and does root cause diagnosis of a given failure, without relying on record and replay.

Triage [67], Giri [58], and DrDebug [69] use dynamic slicing for root cause diagnosis. Triage works for systems running on a single processor and uses custom checkpointing support. DrDebug and Giri assume that failures can be reproduced in-house by record/replay and that one knows the inputs that lead to the failure, respectively. Gist relies on hardware tracking for slice refinement at low performance cost, and does not assume that failures can be reproduced in-house.

Tarantula [28] and Ochiai [1] record all program statements that get executed during failing and successful runs, to perform statistical analysis in the recorded statements for root cause diagnosis. Gist does not record all program statements, and thus it can be used in production runs.

Unlike other low overhead, in-production root cause diagnosis techniques we know of, Gist tracks data flow in addition to control flow. Triage has alluded to the necessity of tracking data flow, but did not implement it because of potentially high overheads [67]. Gist is able to track the data flow with low overhead due to AsT and hardware watchpoints.

Exterminator [48] and Clearview [51] automatically detect and generate patches for certain types of bugs (e.g., memory errors and security exploits). Gist can assist these tools in diagnosing failures for which they can generate patches.

Initial ideas regarding Gist were explored in a recent workshop paper [30]. This submission presents the design in depth, formalizes the algorithms, reports on our implementation using real hardware, evaluates the prototype on real-world systems, and describes insights we gained from the design and implementation effort.

8. Conclusion

In this paper, we describe failure sketching, a technique that provides developers with a high-level explanation of the root cause of a failure. Failure sketches contain statements, values, and statement orders that cause a failure. Failure sketches display differences in key program properties between failing and successful runs.

We describe the design and implementation of Gist, a tool that combines in-house static analysis with cooperative adaptive dynamic analysis to build failure sketches. Gist is effective, accurate, and efficient. All the failure sketches built by Gist in our evaluation point to root causes that developers used when fixing the failure.

Acknowledgments

We are indebted to our shepherd Gustavo Alonso, to the anonymous reviewers, Vikram Adve, Emery Berger, Edouard Bugnion, James Larus, Madan Musuvathi, and to Yanlei Zhao for their insightful feedback and generous help in improving this paper. We thank Andi Kleen for providing the initial version of the Intel PT driver. This work was supported in part by ERC Starting Grant No. 278656 and by gifts from Intel and Baris Kasikci's VMware graduate fellowship.

References

[1] Abreu, R., Zoeteweij, P., and Gemund, A. J. C. v. An evaluation of similarity coefficients for software fault localization. In *PRDC* (2006).

[2] Altekar, G., and Stoica, I. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles* (2009).

[3] Ammons, G., and Larus, J. R. Improving data-flow analysis with path profiles. In *Intl. Conf. on Programming Language Design and Implem.* (1994).

[4] Arulraj, J., Chang, P.-C., Jin, G., and Lu, S. Production-run software failure diagnosis via hardware performance counters. In *ASPLOS* (2013).

[5] Arulraj, J., Jin, G., and Lu, S. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Intl. Conf. on Ar-*

chitectural Support for Programming Languages and Operating Systems (2014).

[6] ARUMUGA NAINAR, P., AND LIBLIT, B. Adaptive bug isolation. In *Intl. Conf. on Software Engineering* (2010).

[7] BARIS KASIKCI, BENJAMIN SCHUBERT, G. C. Gist. http://dslab.epfl.ch/proj/gist/, 2015.

[8] BEEMAN STRONG. Debug and fine-grain profiling with intel processor trace. http://bit.ly/1xMYbIC, 2014.

[9] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO* (2003).

[10] BUTENHOF, D. R. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[11] CHEN, H., YU, J., CHEN, R., ZANG, B., AND YEW, P.-C. Polus: A powerful live updating system. In *ICSE* (2007).

[12] CHILIMBI, T. M., LIBLIT, B., MEHRA, K., NORI, A. V., AND VASWANI, K. HOLMES: Effective statistical debugging via efficient path profiling. In *Intl. Conf. on Software Engineering* (2009).

[13] CHOI, J.-D., AND ZELLER, A. Isolating failure-inducing thread schedules. In *ISSTA* (2002).

[14] DUNLAP, G. W., LUCCHETTI, D., CHEN, P. M., AND FETTERMAN, M. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments* (2008).

[15] FITZPATRICK, B. Memcached. http://memcached.org, 2013.

[16] GILCHRIST, J. Parallel BZIP2. http://compression.ca/pbzip2, 2015.

[17] GLERUM, K., KINSHUMANN, K., GREENBERG, S., AUL, G., ORGOVAN, V., NICHOLS, G., GRANT, D., LOIHLE, G., AND HUNT, G. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles* (2009).

[18] GODEFROID, P., AND NAGAPPAN, N. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification* (2008).

[19] HAUSWIRTH, M., AND CHILIMBI, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS* (2004).

[20] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. ISCA.

[21] Apache httpd. http://httpd.apache.org, 2013.

[22] INTEL. *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 2. 2015.

[23] INTEL CORPORATION. Intel processor trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing, 2013.

[24] Linux branch with intel pt support. https://github.com/virtuoso/linux-perf/tree/intel_pt, 2015.

[25] JIN, G., THAKUR, A., LIBLIT, B., AND LU, S. Instrumentation and sampling strategies for cooperative concurrency bug isolation. *SIGPLAN Not.* (2010).

[26] JIN, G., ZHANG, W., DENG, D., LIBLIT, B., AND LU, S. Automated concurrency-bug fixing. In *OSDI* (2012).

[27] JOHN CRISWELL. The information flow compiler. https://llvm.org/svn/llvm-project/giri/, 2011.

[28] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE* (2005).

[29] KASIKCI, B., BALL, T., CANDEA, G., ERICKSON, J., AND MUSUVATHI, M. Efficient tracing of cold code via bias-free sampling. In *USENIX Annual Technical Conf.* (2014).

[30] KASIKCI, B., PEREIRA, C., POKAM, G., SCHUBERT, B., MUSUVATHI, M., AND CANDEA, G. Failure sketches: A better way to debug. In *Workshop on Hot Topics in Operating Systems* (2015).

[31] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. Data races vs. data race bugs: Telling the difference with Portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2012).

[32] KASIKCI, B., ZAMFIR, C., AND CANDEA, G. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles* (2013).

[33] KENDALL, M. G. A new measure of rank correlation. *Biometrika* (1938).

[34] KLEEN, A. simple-pt linux driver. https://github.com/andikleen/simple-pt, 2015.

[35] LATTNER, C. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[36] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization* (2004).

[37] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable statistical bug isolation. In *Intl. Conf. on Programming Language Design and Implem.* (2005).

[38] LIBLIT, B. R. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, Dec. 2004.

[39] LU, S., PARK, S., SEO, E., AND ZHOU, Y. Learning from mistakes – A comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2008).

[40] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.* (2005).

[41] MACHADO, N., LUCIA, B., AND RODRIGUES, L. Concurrency debugging with differential schedule projections. In *PLDI* (2015).

[42] MANEVICH, R., SRIDHARAN, M., ADAMS, S., DAS, M., AND YANG, Z. PSE: explaining program failures via postmortem static analysis. In *Symp. on the Foundations of Software Eng.* (2004).

[43] MARJAMÃĎKI, D. Cppcheck. http://cppcheck.sourceforge.net/, 2015.

[44] MCCONNELL, S. *Code Complete*. Microsoft Press, 2004.

[45] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The paradyn parallel performance measurement tool. *Computer* (1995).

[46] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution ef?ciently. *ISCA* (2008).

[47] MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[48] NOVARK, G., BERGER, E. D., AND ZORN, B. G. Exterminator: Automatically correcting memory errors with high probability. In *Intl. Conf. on Programming Language Design and Implem.* (2007).

[49] PAPAMARCOS, M. S., AND PATEL, J. H. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA* (1984).

[50] PARK, S., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., LU, S., AND ZHOU, Y. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles* (2009).

[51] PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. Automatically patching errors in deployed software. In *Symp. on Operating Sys. Design and Implem.* (2010).

[52] POKAM, G., PEREIRA, C., HU, S., ADL-TABATABAI, A.-R., GOTTSCHLICH, J., HA, J., AND WU, Y. Coreracer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (2011), MICRO-44, ACM, pp. 216–225.

[53] QIN, F., TUCEK, J., ZHOU, Y., AND SUNDARESAN, J. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems 25*, 3 (2007).

[54] QUORA. What is a coder's worst nightmware? http://www.quora.com/What-is-a-coders-worst-nightmare.

[55] RASTISLAV BODIK, S. A. Path-sensitive value-flow analysis. In *Symp. on Principles of Programming Languages* (1998).

[56] RIJSBERGEN, C. J. V. *Information Retrieval*. Butterworth-Heinemann, 1979.

[57] SADOWSKI, C., AND YI, J. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools* (2014), PLATEAU.

[58] SAHOO, S. K., CRISWELL, J., GEIGLE, C., AND ADVE, V. Using likely invariants for automated software fault localization. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2013).

[59] SLABY, J. Llvm slicer. https://github.com/jirislaby/LLVMSlicer/, 2014.

[60] SQLite. http://www.sqlite.org/, 2013.

[61] STENBERG, D. Curl bug 965. http://sourceforge.net/p/curl/bugs/965/, 2013.

[62] STENBERG, D. Curl. http://curl.haxx.se/, 2015.

[63] STODDARD, B. Apache bug 21287. https://bz.apache.org/bugzilla/show_bug.cgi?id=21287, 2003.

[64] SWEENEY, L. K-Anonymity: A model for protecting privacy. In *Intl. Journal on Uncertainty, Fuzziness and Knowledge-based Systems* (2002).

[65] THE ASSOCIATED PRESS. Northeastern blackout bug. http://www.securityfocus.com/news/8032, 2004.

[66] Transmission. http://www.transmissionbt.com/, 2015.

[67] TUCEK, J., LU, S., HUANG, C., XANTHOS, S., AND ZHOU, Y. Triage: diagnosing production run failures at the user's site. In *Symp. on Operating Systems Principles* (2007).

[68] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Doubleplay: Parallelizing sequential logging and replay. *TOCS 30*, 1 (2012).

[69] WANG, Y., PATIL, H., PEREIRA, C., LUECK, G., GUPTA, R., AND NEAMTIU, I. Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In *CGO* (2014).

[70] WEIDENDORFER, J. Kcachegrind. http://kcachegrind.sourceforge.net/html/Home.html, 2015.

[71] WEINING GU, ZBIGNIEW KALBARCZYK, RAVISHANKAR K. IYER, ZHEN-YU YANG. Characterization of linux kernel behavior under errors, 2003.

[72] WEISER, M. Program slicing. In *Intl. Conf. on Software Engineering* (1981).

[73] WHEELER, D. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[74] WILSON, P. F., DELL, L. D., AND ANDERSON, G. F. *Root Cause Analysis : A Tool for Total Quality Management*. American Society for Quality, 1993.

[75] WU, J., CUI, H., AND YANG, J. Bypassing races in live applications with execution filters. In *Symp. on*

Operating Sys. Design and Implem. (2010).

[76] XIN, B., SUMNER, W. N., AND ZHANG, X. Efficient program execution indexing. In *Intl. Conf. on Programming Language Design and Implem.* (2008).

[77] YU, J., AND NARAYANASAMY, S. A case for an interleaving constrained shared-memory multi-processor. In *Intl. Symp. on Computer Architecture* (2009).

[78] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Symp. on Operating Systems Principles* (2005).

[79] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2010).

[80] ZAMFIR, C., ALTEKAR, G., CANDEA, G., AND STOICA, I. Debug determinism: The sweet spot for replay-based debugging. In *Workshop on Hot Topics in Operating Systems* (2011).

[81] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002).

[82] ZHANG, W., LIM, J., OLICHANDRAN, R., SCHERPELZ, J., JIN, G., LU, S., AND REPS, T. ConSeq: Detecting concurrency bugs through sequential errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2011).