Relational Debugging — Pinpointing Root Causes of Performance Problems

Xiang (Jenny) Ren, Sitao Wang, Zhuqi Jin, David Lion, and Adrian Chiu, *University of Toronto;* Tianyin Xu, *University of Illinois at Urbana-Champaign;* Ding Yuan, *University of Toronto*

https://www.usenix.org/conference/osdi23/presentation/ren

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

Relational Debugging — Pinpointing Root Causes of Performance Problems

Xiang (Jenny) Ren[1], Sitao Wang[1], Zhuqi Jin[1], David Lion[1], Adrian Chiu[1], Tianyin Xu[2], and Ding Yuan[1]

[1]University of Toronto
[2]University of Illinois at Urbana-Champaign

Abstract

Performance debugging is notoriously elusive—real-world performance problems are rarely clear-cut failures, but manifest through the accumulation of fine-grained symptoms. Oftentimes, it is challenging to determine performance anomalies—*absolute* measures are unreliable, as system performance is inherently *relative* to workloads. Existing techniques focus on identifying absolute predicates that deviate between executions, which limits their application to *performance* problems.

This paper introduces *relational debugging*, a new technique that automatically pinpoints the root causes of performance problems. The core idea is to capture and reason about relations between fine-grained runtime events. We show that relations provide immense utilities to explain performance anomalies and locate root causes. Relational debugging is highly effective with a minimal two executions (a good and a bad run), eliminating the pain point of producing and labeling many different executions required by traditional techniques.

We realize relational debugging by developing a practical tool named Perspect. Perspect directly operates on x86 binaries to accommodate real-world diagnosis scenarios. We evaluate Perspect on twelve challenging performance issues with various symptoms in Go runtime, MongoDB, Redis, and Coreutils. Perspect accurately located (or excluded) the root causes of these issues. In particular, we used Perspect to diagnose *two open bugs*, where developers failed to find root causes—the root causes reported by Perspect were confirmed by developers. A controlled user study shows that Perspect can speed up debugging by at least 10.87 times.

1 Introduction

Performance makes or breaks a software system: severe performance problems lead to unresponsiveness and even malfunctions; even seemingly-small performance degradations can incur high costs—a half-second search delay reduces Google's revenue by 20% [33]. Therefore, it is crucial to diagnose performance problems in a timely manner.

Performance debugging is known to be elusive and difficult. Unlike functional failures with clear-cut symptoms, such as crashes and runtime exceptions, performance problems are typically observed via the cumulative effect of fine-grained symptoms over time, such as latency increases due to regressions of code efficiency and resource overuse due to leaks. While fine-grained symptoms can potentially be identified by profilers [9, 10, 12, 13, 19], profiling alone cannot explain a performance anomaly—not every local symptom is related to the anomaly. Causality analysis [34, 37, 42] captures runtime events that are causally related to the symptoms, but it does not pinpoint the root causes in the code; the causality graph can be complex to navigate and analyze. In fact, it can be even challenging to determine whether or not the observed is performance an anomaly, because *absolute* measures are unreliable—system performance is inherently *relative* to inputs and workloads.

Existing performance diagnosis techniques target specific types of root causes and thus are limited when applied to many challenging performance problems. For example, X-ray [15] diagnoses performance anomalies due to unexpected inputs or configuration values by summarizing performance impact of each input/configuration value; however, as a tool designed for end users, X-ray does not address problems rooted in the source code. Statistical debugging [24, 26, 29, 32, 38] can address certain types of performance problems which result in differences in program predicates (e.g., branches and returns) [39]. However, unlike functional failures, many performance problems do not cause changes in predicates (e.g., due to distribution changes in runtime events). Besides, it can be challenging to design predicates and statistical models in the first place [39].

This paper introduces *relational debugging*, a new technique that automatically pinpoints the root causes of performance problems. The core idea is to capture *relations* between fine-grained runtime events. We show that relations provide immense utilities to explain performance anomalies and locate root causes. Relational debugging analogizes performance problems to *relative motion* in physics—just like the speed of an object is a relative measure depending on the reference frame, so is performance when viewed from different runtime events during program execution. Root causes of performance problems can be revealed by analyzing changes of

relative measures of these events (i.e., their relations) between a good run and a bad run (with performance anomalies).

Consider a real-world performance issue (see §2.1), where the developer observes an abnormal increase in memory consumption by a server application. Potential root causes can be: 1) an influx of more requests (in which consuming more memory is normal), 2) each request allocating more memory (indicating regression of code efficiency), and 3) allocated memory not being reclaimed (indicating memory leaks). Each of these hypotheses can be expressed as a relation (a measure relative to an event): 1) the number of requests *relative to* each time epoch, 2) the amount of memory allocated *relative to* each request, 3) the amount of memory reclaimed *relative to* each request. Relational debugging verifies the hypotheses by comparing the three relations in executions with and without the observed performance anomaly. In this example, 1) and 2) are the same, while 3) decreases significantly, suggesting memory leaks. Relational debugging further pinpoints the root cause of the memory leak by analyzing fine-grained relations. It finds that *relative* to all memory objects not reclaimed by the garbage collector (GC), many more are unreachable by pointers in the abnormal execution than the normal execution—a bug in the GC mistakenly treats constant values as pointers.

Relational debugging is highly effective with a minimal two executions (a good and a bad), eliminating the pain point of producing and labeling many different executions required by traditional statistical techniques [24,26,29,32,38,39]. Notably, relational debugging utilizes the repetitiveness of performance symptoms which accumulate during the execution—a single execution offers a large sample of normal or abnormal patterns. Relational debugging is *generic* to performance problems with different types of root causes, including inefficient code, misconfigurations, and workload changes, etc. Moreover, relations can describe different types of symptoms such as slowdowns and memory overuse.

We realize relational debugging by developing a practical tool named Perspect. Perspect is fully *automatic*; it does not require manual instrumentations or annotations. Perspect takes the symptoms (such as a program counter that indicates excessive memory usage or a function with abnormal execution time) as inputs. It outputs the relations that are 1) *causally relevant* to the symptoms and 2) have significant impacts on the performance measures of the symptom; such relations describe the root causes of the performance problems. Perspect directly operates on x86 binaries to accommodate real-world diagnosis scenarios (e.g., when the binary build is nontrivial), and can tolerate small differences in the binaries.

Perspect focuses on capturing a small set of relations that can pinpoint the root cause. Instead of tracking all possible relations of every runtime event, Perspect reduces the search space by identifying runtime events that are causally related to the symptoms through control or data flow. Perspect then filters out relations that are not changed between the good and bad executions. For relations that are changed between the executions, Perspect automatically differentiates between relation changes that reflect the *effect* (e.g., a decrease of reclaimed memory relative to each request), and changes that reflect the *cause* (e.g., an increase in objects not referred by real pointers). These strategies effectively filter out most of the irrelevant relations, with the remaining relations being root cause candidates. Lastly, Perspect ranks root-cause relations based on their impacts on performance measures of the observed symptom, and outputs them in descending order.

Perspect is carefully implemented so its analysis is both *precise* and *scalable* to real complex systems. It has an efficient algorithm that computes all the relations by traversing the dependency graph only once. In addition, it distributes the precise but expensive data-flow dependency analysis onto different servers. Finally, Perspect is able to handle the difference between two different versions of the binary executables.

We evaluate Perspect on twelve real-world performance issues from complex systems (Go runtime, MongoDB, Redis, and Coreutils), covering different symptoms (slowdown and memory overuse). Perspect effectively locates the root causes of these challenging issues. Notably, we applied Perspect to *two open issues* where developers failed to find the root causes; Perspect successfully located the root causes of both issues which are confirmed by the developers. For an issue where the root cause is located outside the target program (in the OS kernel), which took developers a long time to debug, Perspect correctly excluded the root cause from the application code, since it detects no significant relation changes.

In summary, this paper makes the following contributions:

- We present relational debugging, a new technique that analyzes the relations between causally related events, seizing the essence of performance debugging.
- We build Perspect, a practical tool that realizes relational debugging for large, complex real-world systems. Perspect directly operates on x86 binaries and accommodates real-world diagnosis/debugging scenarios.
- We show that Perspect can effectively locate the root causes of real performance problems, and can help resolve two previously unresolved issues. The source code of Perspect and the dataset are available at `https://gitlab.dsrg.utoronto.ca/dsrg/perspect`.

2 Relational Debugging by Examples

We use two real-world examples to show how relational debugging locates the root causes of challenging performance problems in complex software systems. Both problems are among the most challenging performance issues faced by developers, who were unable to locate the root causes with existing tools. Specifically, the Go runtime bug (§2.1) took a year of investigation, and the MongoDB bug (§2.2) is an open issue that developers failed to diagnose. Perspect automatically pinpoints the root causes in the form of relations.

2.1 Go-909: A Memory Leak

Go-909 is among the most famous performance bugs in the Go runtime. The developers reported that "*garbage collection is ineffective on 32-bit*" systems, causing workloads to run out of memory [2]. The same bug resulted in 9 other tickets (which turned out to have the same root cause) and at least 2 extensive discussion threads on Golang's email list.The bug was also discussed in Hacker News with 147 comments [4].

2.1.1 Challenges of Debugging Go-909

Debugging Go-909 was very challenging not only for application developers but also for developers of the Go runtime. During the course, many wrong hypotheses, some of which were wildly off, were developed. For example, a developer believed that the bug was caused by the Go runtime forgetting to `munmap` freed memory [1]. There are at least three other bugs, of which developers could not agree on the root cause, that were eventually attributed to Go-909. After more than a year of investigations, the root cause was discovered through a trial-and-error process: the bug can be worked around by commenting out specific packages that contain a lot of static constants.

Existing performance debugging techniques can hardly address Go-909. First, Go-909 does not always cause a clear-cut out-of-memory error; in fact, many developers reported the bug simply after noticing their programs using more memory than expected [1, 3]. Moreover, since the root cause is not in program inputs, isolating faulty inputs using X-ray [15] or delta debugging [48] does not help. The root cause also can hardly be revealed by statistical debugging [32, 39], because it does not manifest in any abnormal predicates such as branch targets, unexpected return values, or scalar-pairs [39]. In fact, the memory leak also occurred in the reference executions (64-bit systems), only affecting many fewer objects.

2.1.2 Root Cause

Figure 1 shows the simplified code snippet in the buggy version of the Go runtime. Go programs invoke `runtime.malloc` to allocate memory and the Go runtime uses a mark-and-sweep garbage collector (GC). Once an object is allocated (L2), `runtime.malloc` increments the `heap_size` counter (L3).

The `mark` function looks for objects that are reachable through variables on the stack and in the data segments. Unmarked objects will later be reclaimed by `sweep`. During the stack scan, `mark` takes the pointer to the start of the stack and data segments (`b`), as well as the size of the respective regions (`n`). For every word on the stack and data segments, it initially assumes it to be a pointer and checks whether it points to an address inside the heap's range (L15). If so, `mark` sets the "marked" bit in the metadata of the object (L18–19). Then, `mark` uses an iterative worklist `w` to further scan the memory based on the marked pointers. Later, `sweep` goes through each span, a memory region containing same-sized blocks. The

```
1  void* runtime.malloc(unintptr size, ...) {
2    void *p = runtime.Alloc(...);
3    heap_size += size;
4    uintptr bits = get_metadata(p);
5    ...
6    set_metadata(p, bits);
7    return p;
8  }
9  // Mark objects reached by pointers
10 static void mark(byte *b, int64 n) {
11   void **w = get_buffer_head();
12   while(b != nil) { ...
13     for(i = 0; i < n; i++) {
14       byte *p = (byte*)b[i];
15       if(p < HEAD_START || p >= HEAD_USED)
16         continue;
17       uintptr bits = get_metadata(p);
18       bits |= BIT_MARKED; /* set mark bits */
19       set_metadata(p, bits);
20       *w++ = p;
21     }
22     b = *--w;
23     n = get_size(b);
24   }
25 }
26 // Reclaim unreachable objects
27 static void sweep(void) {
28   uintptr size = getsize(span);
29   for(byte *p = span->start; ... p += size) {
30     uintptr bits = get_metadata(p);
31     if((bits & BIT_MARKED) != 0) {
32       bits &= ~BIT_MARKED; /* clear mark bit */
33       continue;
34     }
35     set_metadata(p, bits);
36     runtime.Free(p, size, ...);
37     heap_size -= size;
38   }
39 }
```

GC log: heap_size is logged

GC log: heap_size is logged

```
./Perspect run_64 run_32
  run_64: R<(L7|L18) = 0.99 // 64-bit (good run)
  run_32: R<(L7|L18) = 0.01 // 32-bit (bad run)
```

Figure 1: Code snippets showing how Perspect locates the root causes of Go-909 by pinpointing the changed relation between L7 and L18 by comparing the two runs.

loop at L29 goes through each block, checks if the marked bit is set, and if so, clears the mark bit (L32) and continue on to the next block. Otherwise, it frees the object and decrements the heap size (L37).

The implementation of `mark` suffers from fake pointers—non-pointer variables that happen to have values within the range of `HEAP_START` and `HEAP_USED` (L15). The objects pointed to by those variables will not be reclaimed. [1] The defect affects both 32- and 64-bit systems; however, fake pointers occur orders of magnitude more frequently in 32-bit systems than 64-bit systems due to data layouts differences.

[1] This is a known side effect of using a conservative garbage collector.

2.1.3 Relational Debugging Go-909

Perspect takes as inputs a good run (which uses the 32-bit Go runtime) and a bad run (which uses the 64-bit Go runtime) of the Go program provided by the bug reporter, as well as the symptom. Since the bug manifests in abnormal heap sizes in the GC log, we (users) feed Perspect the `heap_size` variable which records the heap size value printed in the log. Perspect identifies the instructions that modify `heap_size` , i.e., L3 and L37 in the code of Figure 1, and Perspect treats these instructions as symptom instructions. Perspect will not only analyze what causes these symptom instructions to execute, but also what prevents these symptom instructions from executing; To do this, Perspect also identifies "negation" symptoms which are instructions that directly prevent a symptom instruction from executing, for example, L18, because each time L18 executes which marks and object, it directly prevents an instance of L37 which reclaims the object.

Perspect carries out relational debugging starting from instructions that directly determine the `heap_size` (L3, L37, and L18). It builds relations between symptom instructions and their causal predecessors. In this case, Perspect efficiently locates the root cause to a single relation (see Table 1 for notations):

$$R\blacktriangleleft(\mathrm{L7}_{malloc.return} \mid \mathrm{L18}_{mark})$$

On 64-bit systems, the relation is expected to be $1:1$, indicating that for every marked object on L18, there exists a dependency on a pointer returned by `malloc`. Yet, on 32-bit systems, the relation drops to $1:0.01$, i.e., only 1% of the marked objects have a pointer returned by `malloc`. The remainings are pointed to by fake pointers (constant values).

Note that the $1:1$ relation in the reference run on 64-bit systems is *not* an invariant. Precisely, Perspect observed the relation to be $1:0.99$, i.e., 99% of the marked objects are pointed to by a pointer returned by `malloc`. This is because the defect still exists in 64-bit systems, but only affecting 1% of the objects in the reference run.

$R\blacktriangleleft(\mathrm{L7|L18})$ is not the only relation built by Perspect. Taking L18 as an example, Perspect builds four relations w.r.t L18's causal predecessors $L1$ and $L10$:

- $R\blacktriangleleft(L1|L18)$: the distribution of the number of marked objects that depend on `malloc`;
- $R\blacktriangleright(L18|L1)$: the distribution of the number of times an object (still reachable by real pointers) gets marked;
- $R\blacktriangleleft(L10|L18)$: the distribution of the number of marked objects that depend on `mark`;
- $R\blacktriangleright(L18|L10)$: the distribution of the number of objects marked per `mark` call;

Perspect filters out $R\blacktriangleright(L18|L1)$ because the distribution of the lifetimes of objects reachable by real pointers do not change significantly between the good and bad run; and Perspect filters out $R\blacktriangleleft(L10|L18)$ because each marked object always depend on one invocation of `mark`. $R\blacktriangleright(L18|L10)$ is changed across the runs, because fake pointers causes many more objects to be marked during each `mark` call in the bad run, but Perspect also excludes it because relational debugging recognizes that the relation only reflects the *effect* of the root cause, but is not the root cause. Finally, for $R\blacktriangleleft(L1|L18)$, Perspect refines it to the most specific variant, $R\blacktriangleleft(L7|L18)$. The other relations (e.g., those w.r.t symptom instructions at L3 and L37) are handled in similar ways, and eventually filtered out. We discuss Perspect's filtering and refinement techniques in §3.3.

Ln	An static instruction at line n
$e\mathrm{L}n_i$	The i-th instance of Ln in the execution
$\mathcal{S}$	A symptom instruction
$\mathrm{e}\mathcal{S}_i$	A symptom event
P	A static insn. & causal predecessor of $\mathcal{S}$
P^+	A static insn. & direct causal successor of P
$R\blacktriangleright(\mathcal{S}\|P)$	A forward relation between P and $\mathcal{S}$
$R\blacktriangleleft(P\|\mathcal{S})$	A backward relation between P and $\mathcal{S}$
$R\blacktriangleleft\blacktriangleright(P,\mathcal{S})$	A pair of forward and backward relations
$R_?(P,\mathcal{S})$	A relation btw. P and $\mathcal{S}$ of unspecified direction

Table 1: Notations for relations.

2.2 MongoDB-57221: A Slowdown

> *"[Perspect's result] ties all the pieces together into a nice explanation. That explanation being, having some unnecessary cursors simply open on failed plans isn't strictly the problem. It's that we're paying the (also unnecessary) cost to reposition them after every delete + restore."*
> —MongoDB developer's comment on Perspect's result.

MongoDB-57221 is an open bug which developers were unable to diagnose. It is triggered by executing a query that deletes all the records in the table. The query could slow down by 5x on the buggy version. During the deletion, MongoDB uses a cursor, i.e., a pointer to a record in the table that indicates the current position, to locate each record. It advances the cursor to the next record after deleting the previous one; this process is known as cursor restoration.

This bug is caused by maintaining unnecessary cursors on multiple query plans. Before the query execution, MongoDB generates multiple query plans, performs a sandboxed trial of these plans, and chooses the best-performing plan. Different query plans use different indexes, thereby deleting records in different orders. The actual order of the deletion is determined by the index of the *winning* plan. However, MongoDB still keeps the rejected plans and their cursors. More importantly, it restores the cursor of each rejected plan following the same order as the winning plan. Whereas for the winning plan, restoring the cursor means simply moving to the next position, for the losing plan, restoring the cursor requires traversing through many already deleted records. And if the number of deleted records encountered exceeds a threshold, it flags the page for eviction. The increase in unnecessary evictions leads to the slowdown.

Developers were unable to understand the root cause of this bug, despite them quickly identifying evictions being the bottleneck based on profiling and being aware of the existence of multiple query plans. However, they could not explain *why* excessive evictions occurred, because they could not establish the causal link between evictions and the cursor restoration of the rejected plans. This led to rounds of ping-pongs between the Storage Engine team (responsible for the eviction) and the Query Execution team (responsible for maintaining multiple plans). The storage team suspected that the slowdown was caused by maintaining multiple plans, but the query execution developers believed that it was cheap to keep multiple plans around. And they further suspected that the slowdown was caused by the threshold misconfiguration that triggered the eviction. In the end, seven different developers actively discussed this issue for over a month. The JIRA discussion has over 3,000 words in 18 comments, with multiple rounds of reproduction and profiling with multiple screenshots posted. And the two teams had multiple off-line teleconference discussions. Still, they were unsure why the slowdown occurred.

Perspect pinpoints the root cause and explains the slowdown. It captures that the costly evictions are causally dependent on the restoration of multiple cursors. Figure 2 displays a simplified version of static dependency graph for eviction. Starting from `eviction` as the symptom, Perspect returns the root cause candidate: (1) $R\triangleright$(`cursor_search` | `restore`), where `restore` invokes `cursor_search` once in the good run to restore one cursor, but twice in the bad run to restore two cursors. Perspect infers that restoring an additional cursor causes a significant increase in evictions in the bad run.

Moreover, Perspect specifically infers that during cursor restoration, additional traversals through dead records increased evictions. It returns (2) $R\triangleleft\triangleright$(`eviction` | `search_forward`) as a new pair of relations unique to the bad run: `search_forward` is invoked by `cursor_search` to search for the next cursor position by traversing forward in the records. In the good run, `search_forward` almost always locates the next cursor position immediately, triggering no evictions; whereas in the bad run, `search_forward` traverses through many dead records and triggers additional evictions. Perspect also returns (3) $R\triangleright$(`search_backward` | `cursor_search`) as a root cause candidate. In the good run, `cursor_search` invokes `search_backward` only 1% of the time, because `search_forward` locates the next cursor position most of the time; however, in the bad run, `cursor_search` invokes `search_backward` half of the time. The increased searches lead to additional evictions.

3 Perspect

Generally speaking, debugging a performance problem takes three steps: 1) observing symptom(s), 2) capturing runtime events that are causally related to the symptom(s), and 3) locating the root cause. Perspect automates the last two steps, taking the symptoms as its inputs. Perspect supports different

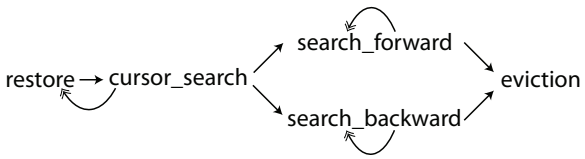

Figure 2: A simplified version of the static dependency graph for eviction. Each edge with a single arrow represents a dependency. An edge with a double arrow represents a backedge in a loop. `restore` loops through every cursor and restores each by invoking `cursor_search`. `cursor_search` then invokes `search_forward` which looks for the next record by iterating forward. If `search_forward` returns without locating the next record, `cursor_search` will then invoke `search_backward`. If `search_backward` or `search_forward` encounters too many dead records, it will trigger `eviction`.

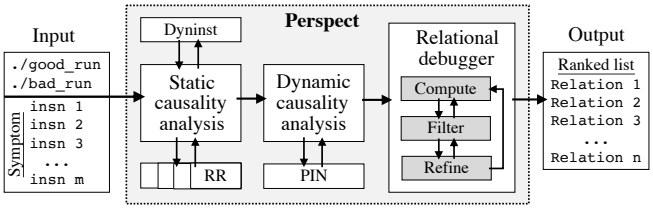

Figure 3: An overview of Perspect's workflow

forms of symptoms, including: program variables that record the symptoms (e.g., `heap_size` in Go-909), slow functions (such as `eviction` in MongoDB-57221), and basic blocks (captured by profilers like `gprof` [12]). Perspect automatically identifies the instructions related to the input symptoms as the starting points of its analysis (§3.1). Perspect outputs a list of relations that explain the root cause in descending order based on their impacts on the observed symptom.

Figure 3 shows the workflow of Perspect. Perspect uses causality analysis to reduce the search space of relational debugging to a small set of instructions and their runtime instances that are causally related to the symptom (see §3.2). Perspect then performs relational debugging to build relations with regard to the symptom. It filters out relations that are irrelevant to the symptom, refines relations to be specific to the root cause, and ranks relations based on their impacts on the observed symptom (see §3.3).

3.1 Bootstrapping with Symptoms

Perspect bootstraps itself by identifying the instructions that reflect the observed symptoms. If the symptom is a performance counter recorded in a program variable (such as `heap_size` in Go-909), Perspect identifies the instructions that use the variable as an operand. If the symptom is a function or a basic block (typically captured by profilers), Perspect identifies the first instruction of the function or the first instruction in the basic block. Hence, Perspect converts different types of symptom inputs to unified starting points in the form

of instructions, termed *symptom instructions*, denoted as $\mathcal{S}$ in Table 1. A dynamic instance of the symptom instruction is a *symptom event*.

Each symptom instruction is assigned a *weight*. The weight can be the value of a variable in an instruction, such as `size` in L3 or L37 in Go-909 (Figure 1), or the estimated time or cycles taken by a code block (e.g., `eviction`).

Perspect also identifies instructions that prevent the occurrence of symptoms (i.e., the negation of a symptom) as a special type of symptom instructions. An example is L18 in Go-909. Perspect searches every conditional branch that dominates a symptom instruction, which could prevent the symptom from occurring (e.g., L31 in Go-909 w.r.t L37). It then identifies the instructions that determine the branch conditions (e.g., L18 in Go-909). In practice, we find it suffices to only include instructions of negation for the initial symptom instructions. Therefore, in our current implementation, Perspect does not recursively search for negation symptoms.

3.2 Causality Analysis

Perspect reduces the search space of relational debugging by restricting the subsequent analysis to a small subset of instructions and their runtime instances that are causally related to the symptoms. The high-level idea is to dynamically track instructions that the symptoms are causally dependent on through control- and data-flow (aka information flow) during the execution of the good or bad reproduction runs. Specifically, Perspect generates a *dynamic* program dependency graph that contains instances of instructions that the symptom is causally dependent on.

The causality tracking is done in two phases. Perspect first generates the *static* program dependency graph (SDG) [25] for all the symptom instructions from the program. In the SDG, a node v is an instruction and an edge (u, v) represents a causal dependence, either a data dependence (a data value v depends on) or a control dependence (a control condition on which v depends on). We call u a *causal predecessor* of v and v a *causal successor* of u. To generate the SDG, Perspect performs *backward* causality tracking: it starts from each symptom instruction (including negation symptoms) and recursively includes causal predecessor instructions by tracking control or data flow.

Perspect then automatically instruments the instructions in the program binary that belong to the SDG; it later generates *dynamic* program dependency graphs (DDGs) by running the program binary and monitoring the execution of each instrumented instruction. Different from the SDG, which consists of static instructions, in a DDG, a node is a runtime *event*—an instance of an instruction in the execution. Each instruction in the SDG can correspond to multiple events in a DDG. We use $e\mathrm{L}n_i$ to denote an event of the i-th occurrence of the instruction at line n (i.e., Ln) in the execution (see Table 1).

Section 4 describes the implementation details.

3.3 Relational Debugger

Within the scope of instructions that are causally related to the symptom(s), Perspect computes the relations between the symptom instructions and their causal predecessors in the SDG, based on runtime dependencies derived from the DDGs (§3.3.1). Perspect only considers relations that are changed between the good and the bad executions as potential root causes by filtering out unchanged relations (§3.3.2). Perspect further refines each relation until it finds the specific relation that captures a root cause of the change in the number of symptom events between the good and the bad executions (§3.3.3). The filtering and refinement steps are iterated repeatedly to select a minimal set of relations as the candidates of the root cause (Figure 3). Lastly, Perspect ranks the root-cause candidate relations based on their impact on the symptoms (§3.3.4).

We use Go-909 (Figure 1) as a running example when explaining the above components.

3.3.1 Computing Relations

For each symptom (including the negative symptoms), Perspect computes the relation between an instruction P, which the symptom depends on causally, and the corresponding symptom instruction $\mathcal{S}$. Both P and $\mathcal{S}$ are nodes in the SDG generated in §3.2. The relation is computed based on the DDG (§3.2) which records runtime events of P and $\mathcal{S}$ during the executions. Perspect computes relations for the good run and the bad run, respectively.

Perspect starts by only considering the relation between $\mathcal{S}$ and the root nodes of the SDG as P. These root nodes are typically the entry point of a software module and the `main` function. It gradually considers other events on the causal dependency paths between the root node and $\mathcal{S}$ using a refinement process described in §3.3.3.

Perspect computes both *forward relations* and *backward relations*. A forward relation is defined as $R \blacktriangleright (\mathcal{S}|P) = \{n_i\}$, where each element n_i in the set, which corresponds to an instance of instruction P (denoted as eP_i) in the DDG, is the number of causally dependent $\mathcal{S}$ instances ($e\mathcal{S}_j$, $e\mathcal{S}_k$ ... $e\mathcal{S}_m$) of eP_i. Therefore, a relation can be viewed as a distribution; We use the mean of the distribution to represent a relation for simplicity. Here, P can be thought of as serving as a reference point, and $\mathcal{S}$ as the object under observation.

For example, in Go-909, for the symptom instruction $L18_{mark}$ (marking one object), Perspect constructs a relation $R \blacktriangleright (L18_{mark}|L1_{malloc.start})$, which represents the number of times each allocated object got marked. If the first allocated object gets marked (i.e., it results in an instance of $L18$) but the second one does not, then $R \blacktriangleright (L18_{mark}|L1_{malloc.start})$ would be $\{1, 0\}$. In practice, $R \blacktriangleright (L18|L1)$ has a much larger sample size, because hundreds of objects are allocated and marked.

A backward relation is defined as $R \blacktriangleleft (P|\mathcal{S}) = \{m_i\}$, where each element m_i, which corresponds to an instance of $\mathcal{S}$ in the DDG (eS_i), is the number of causally dependent P instances

(eP_i, P_j ... eP_k) of $e\mathcal{S}_i$. Opposite to a forward relation, for a backward relation, the symptom serves as the reference point, and the predecessor as the object under observation.

Regarding the example, $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$, which contains, for each marked object ($L18_{mark}$), the number of causally connected `malloc` instances ($L1_{malloc.start}$); each object pointed to by real pointers is connected to 1 instance of $L1_{malloc.start}$ whereas an object pointed to only by fake pointers is connected to 0 instances.

Note that a backward and forward relation, $R\blacktriangleleft(P|\mathcal{S})$ and $R\blacktriangleright(\mathcal{S}|P)$, complement each other. A forward relation tells: "given the same unit of input, is the same number of symptom events produced?", whereas a backward relation tells: "given the same symptom event, is it still produced by the same units of input?" In Go-909, fake pointers introduce additional causal paths through which the symptom at $L18$ (marking one object) may occur. This is reflected in a change of the backward relation $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$, from 100% to 1% on average; the forward relation $R\blacktriangleright(L18_{mark}|L1_{malloc.start})$, reflecting the number of times each object (reachable from real pointers) gets marked, does not change significantly.

3.3.2 Filtering Unchanged Relations

Perspect filters out a relation $R_?(P,\mathcal{S})$ if it has not changed between the executions of the good and bad runs. Perspect determines if a relation has changed based on its distribution using the two-sample Kolmogorov-Smirnov test [27], with a confidence interval of 95%. For example, in Go-909, the relation $R\blacktriangleright(L18_{mark}|L1_{malloc.start})$ does not change, because, for the objects still reachable from real pointers, the distribution of object life-spans (the number of times they get marked) does not change significantly; therefore, Perspect filters out this relation.

Furthermore, if a relation $R_?(P,\mathcal{S})$ is unchanged across two executions, it implies that the relations between any of P's causal successors—Q—and $\mathcal{S}$ have not changed. Perspect skips the computation of these relations. In other words, if there exists a causal successor Q where $R_?(Q,\mathcal{S})$ is changed, then $R_?(P,\mathcal{S})$ would be changed. Intuitively, it means that the same set of runtime events produces the same symptom events (forward relation) or the same set of symptom events is still produced by the same events (backward relation). This optimization allows us to skip many unnecessary relation computations.

In Go-909, Perspect filters out most of the relations at this step, and only keeps three relations (which will be further refined and filtered in §3.3.3):

- $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$: the number of marked objects reachable from real pointers decreased;
- $R\blacktriangleright(L18_{mark}|L10_{mark.start})$: the number of objects marked per `mark` call increased;
- $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$: the number of objects reclaimed at $L37$ per `sweep` call ($L27$) decreased.

3.3.3 Relation Refinement

Perspect further refines the relations to replace a more "general" relation with a more "specific" one. Refinement is analogous to moving the reference point closer to the object under observation in relative motion. If a relation $R_?(P,\mathcal{S})$ is deemed *refinable*, Perspect replaces the relation with its child relations: $R_?(P_0^+,\mathcal{S})$, $R_?(P_1^+,\mathcal{S})$ ... $R_?(P_n^+,\mathcal{S})$, where $\{P_0^+,P_1^+...P_n^+\}$ are the direct causal successors of P (i.e., children of P). Perspect iteratively refines a relation until it is no longer refinable or can be filtered out by §3.3.2.

Refinement aims to pinpoint the root cause(s). Without refinement, Perspect only outputs relations between $\mathcal{S}$ and root nodes R in the SDG, where R can be the entry point of a module or the `main` function. But the root cause(s) are often located at events on the causal paths connecting R and $\mathcal{S}$. Intuitively, the root cause are events which, if executed, will inevitably cause the performance bug to manifest [50]. The refinement process aims to locate such events.[2]

We design the following two refinement rules:

Rule 1: *A relation $R_?(P,\mathcal{S})$ is refinable, if there is no change in any of the relations between P and its children $\{P_0^+,P_1^+...P_n^+\}$: $R_?(P,P_0^+)$, $R_?(P,P_1^+)$, and $R_?(P,P_n^+)$.*

Intuitively, this rule says P is *not* a root cause; the root cause(s) is located further down the causal paths. Recall that the root cause(s) are events which, once executed, the performance bug will inevitably manifest. But now we have $P+$ that occurred after P in *both* the good and bad run, and $R_?(P,P+)$ does not change. This means that after P executes, the performance bug may still be avoided when $P+$ executes. So we should move one step forward on the causal chain to consider whether $P+$ is the root cause.

With this rule, Perspect refines $R\blacktriangleleft(L1_{malloc.start}|L18_{mark})$ to $R\blacktriangleleft(L7_{malloc.return}|L18_{mark})$ in Go-909, because $R\blacktriangleleft(L1|L7)$ is an invariant that does not change across executions. In the actual code, the program logic between $L1$ and $L7$ is complex; ruling out $L1$ and narrowing it down to $L7$ significantly helps the developer to understand the root cause.

Figure 4 further shows the sequence of refinements performed on $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$. Based on rule 1 we can refine it twice to $R\blacktriangleright(L37|L31)$, because neither $R\blacktriangleright(L29|L27)$ nor $R\blacktriangleright(L31|L29)$ changes.

Even if a relation is not deemed refinable by rule 1, we do not give up—it can still be refined based on rule 2:

Rule 2: *Even if there is a change in $R_?(P,P_i^+)$, $R_?(P,\mathcal{S})$ is still refinable if the change in $R_?(P,P_i^+)$ is caused by the change of $R_?(P',P)$, where P' is a predecessor of P+ and $P' \neq P$.*

Rule 2 differentiates whether a changed relation is a true *root cause*, or merely the *effect* (i.e., manifestation) of the root

[2] Zhang *et al.* defines the root cause as the *inflection point*: if we model the execution as a sequence of instructions, the inflection point in a failure execution F is the point of divergence with a non-failure execution N where N is the non-failure execution that has the longest common prefix with F [50]. Perspect's refinement essentially locates such inflection points.

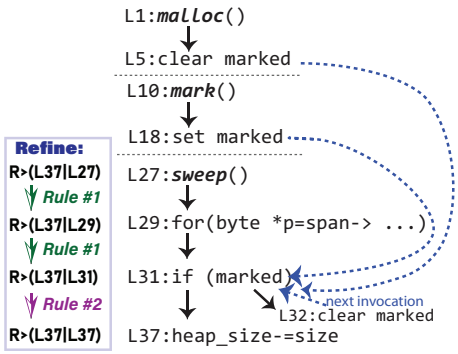

Figure 4: An example to illustrate the refinement rules on Go-909. On the right is (part of) the SDG; a solid edge indicates a control-flow dependency, whereas dotted edges represent dataflow dependencies.

cause. In the former case, we should not be able to find such a P', whereas in the latter case, we can.

Specifically, we consider two cases in our implementation. The first is when P is a branch instruction, P^+ is an instruction in the branch target, and P' is the dataflow direct predecessor of P that defines the branch condition variable. In this case, the change in $R\blacktriangleright(P^+|P)$ is the effect of the change of $R\blacktriangleleft(P'|P)$, which affects the branch direction.

Consider $R\blacktriangleright(L37_{sweep}|L31_{sweep.test})$. Here, P and $P+$ are $L31$ and $L37$, respectively. This relation decreased in the bad run since fewer objects are deemed reclaimable by $L31$, and is therefore no longer refinable according to Rule 1, as illustrated by Figure 4. However, $L31$'s direct dataflow predecessors include $L18$, which sets the mark bit ($L18$ is the P' in this case). The decrease in $R\blacktriangleright(L37|L31)$ is merely caused by the increase in $R\blacktriangleleft(L18|L31)$, i.e., more objects are being marked at $L18$ before $L31$ checks the marked bit. Therefore, according to Rule 2, $R\blacktriangleright(L37|L31)$ is still refinable, and we refine it to $R\blacktriangleright(L37|L37)$ (because $L37$ is $L31$'s direct successor). It can be subsequently filtered based on §3.3.2 since a relation between two identical events doesn't change between runs.This is shown in Figure 4.

Note that we do not need to compute relations on this newly discovered P' separately, because our algorithm guarantees that this relation is computed through other causal paths from the root. For example, after Perspect found $L18$ is the P' in the above example, it does not go on to compute relations between $L18$ and its predecessors, because these relations are already computed through the causal path starting from `mark`.

The second case involves loops, when $P+$ is a loop head and P' is the loop tail. Consider $R\blacktriangleright(L12_{mark.loop}|L10_{mark.start})$. In this case, P is $L10$ and $P+$ is $L12$ (which is a loop head). This relation increased in the bad run because more objects are getting marked. However, this is caused by $L12$'s backedge from $L24$ (loop tail, which is P') executing more often, i.e., $R\blacktriangleright(L24|L10)$ also increased by the same amount.

As a result, even though $R\blacktriangleright(L12_{mark.loop}|L10_{mark.start})$ has changed, $R\blacktriangleright(L18_{mark}|L10)$ can be further refined to $R\blacktriangleright(L18|L12)$. Eventually, $R\blacktriangleright(L18|L12)$ will be filtered out because by further analyzing the dataflow predecessor of $L12$ under Rule 2, Perspect finds that the number of times $L12$ executes is controlled by the size of `w`, which in turn is dataflow-dependent on $L18$ itself (i.e., each time an object is marked, it is pushed onto the queue `w` and popped from the queue later into `b` so `mark` can further scan the content of the object for more pointers). So the relation is refined to $R\blacktriangleright(L18|L18)$ eventually.

By applying the two refinement rules iteratively, Perspect filters both $R\blacktriangleright(L37_{sweep}|L27_{sweep.start})$ and $R\blacktriangleright(L18_{mark}|L10_{mark.start})$. Therefore, Perspect only reports one relation at the end of the filter-refine iterations: $R\blacktriangleleft(L7_{malloc.return}|L18_{mark})$.

3.3.4 Ranking Root-Cause Candidates

After the iterative compute-filter-refine process, the remaining relations are the ones that have not been filtered and are not refinable anymore. We call them root cause candidates.

Perspect ranks the root-cause candidates based on their estimated contributions to performance, in terms of the difference in performance *relative* to the predecessor P. Specifically, for a forward relation $R\blacktriangleright(\mathcal{S}|P) = \{n_i\}$, where each n_i is the number of symptom instances that causally depend on eP_i (the i-th instance of P), Perspect computes a weighted sum: $\sum w_i \times n_i$, where w_i is the *average* weight of the n_i symptom events; $\sum w'_i \times n'_i$ is the weighted sum for the good run. Then the contribution to performance is estimated by $\sum w_i \times n_i - (\sum w'_i \times n'_i) \times \frac{c_P}{c'_P}$, where c_P and c'_P are the number of times P occurred in the bad and good run, respectively. Note that Perspect normalizes $\sum w'_i \times n'_i$ with c_P/c'_P to obtain the performance relative to P in scenarios where the number of times P occurred has changed between the executions. (Say the change in P's occurrences is caused by relation $R\blacktriangleright(\mathcal{S}|P')$, where P' is a predecessor of P, the normalization helps correctly attribute performance impact between $R\blacktriangleright(\mathcal{S}|P)$ and $R\blacktriangleright(\mathcal{S}|P')$.)

In a backward relation $R\blacktriangleleft(P|\mathcal{S}) = \{m_i\}$, Perspect computes weighted sums: $\sum w_i$, $\sum w_j$ where w_i is the weight of the i-th instance of the symptom, and w_j is the weight of the j-th instance of P that can reach a symptom event; And $\sum w'_i$, $\sum w'_j$ are the weighted sums for the good run. Then the contribution to performance is estimated by $\sum w_i - \sum w_j/(\sum w'_i/\sum w'_j)$, where $\sum w_j/(\sum w'_i/\sum w'_j)$ estimates the total number of symptom events, had the same number of symptom events been reachable from P instances in the good run; This formula also handles when the total number of reachable P instances from the symptom differs in the two executions. If the symptom has a negative polarity, as in the case of $L37_{sweep}$, which reduces the heap size as opposed to increasing it, Perspect multiplies its performance impact with -1.

4 Implementation

Perspect is implemented in 10,199 lines of C++ and 14,006 lines of Python. It is built on top of three tools, Dyninst [8] (a binary-level static analysis tool), RR [7] (a deterministic record-and-replay tool), and PIN [11] (a binary instrumentation tool). Perspect operates on application binaries directly.

A key challenge in our implementation is to scale Perspect to the real, complex systems software. This section describes a number of techniques we use for scalability.

4.1 Building Static Dependency Graph (SDG)

Perspect generates the SDG by recursively identifying instructions that are causal predecessors of the symptom instructions via control and data flow. (Figure 4 shows a snippet of the SDG on Go-909.) This is done by three components: 1) a *static analysis* (SA) process running Dyninst, 2) 64 *dynamic dataflow analysis* (DDA) processes running RR (across 4 servers), and 3) a controller. These components form a distributed system that parallelizes computation to scale to real-world systems.

The SA process iteratively infers the instructions on which the symptom instruction $\mathcal{S}$ is control-flow dependent. It analyzes the control-flow graph provided by Dyninst, and only keeps those instructions that $\mathcal{S}$ actually depends on. This analysis is first performed in the function (`f`) that contains $\mathcal{S}$; it is then repeated iteratively in the caller functions by tracing the call-sites starting from `f`.

To obtain dataflow dependencies, Perspect uses a combination of static and dynamic analysis. Perspect only uses Dyninst to obtain the dataflow dependencies of local variables stored in registers or on the stack with static offsets. On the other hand, when a variable is read from other memory locations, *i.e.* the heap or stack locations with non-static offsets, Perspect does not analyze them statically through pointer analysis, because precise pointer analysis can be hard to scale [31]. Instead, Perspect uses the DDA processes to dynamically identify such data dependencies in parallel.

For example, say $\mathcal{S}$ is dominated by an if statement: `if (*p || *q)`; at this point, Perspect needs to infer the dataflow of both `*p` and `*p`, and Dyninst cannot infer the source of the dataflow precisely. Therefore, the SA process sends this request to the controller, which forwards it to a (pre-forked) DDA process to run the RR-guided reproduction. The DDA process first sets breakpoints at the `if` statement to determine the addresses of `*p` and `*q`. It then sets watchpoints at these two addresses and re-run the RR-guided reproduction. [3] (Since execution through RR is deterministic, addresses stay the same across multiple runs.) And via the watchpoints, Perspect locates the store instructions that defined `*p` and `*q`. The DDA process then sends these newly located store instructions back to the SA process (via the controller). This causes the SA process to restart the analysis with these two instructions as the new starting points.

[3] If a breakpoint or watchpoint is not hit in the RR-guided reproduction, Perspect will deem them causally irrelevant to the symptom events and ignore them.

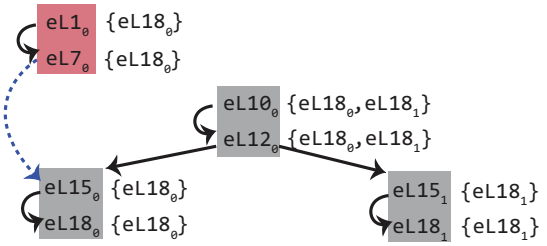

Figure 5: A simplified version of the Dynamic Dependency Graph (DDG) for the symptom instruction at L18 from Go-909. The *red* colour represents the `malloc` function, and the *grey* colour represents the `mark` function. Solid and dotted edges represent control and data flow. The set next to each event is the $\mathcal{S}$-set.

In practice, the SA is orders of magnitude faster than the DDA. Yet, the DDA can be parallelized: for example, the analysis of the dataflow source of `*p` and `*q` can be done in parallel. We create 64 DDA processes, each of which can set at most 4 watchpoints in each run (limited by the number of hardware watchpoints).

4.2 Building Relations

Once the SDG is obtained, Perspect instruments the program at each instruction in the SDG using PIN, and runs the instrumented program to obtain a trace of the good and the bad run, respectively. Perspect builds one DDG for each unique symptom instruction. Each vertex in the DDG is an event, and an edge is a control- or data-flow dependency. Figure 5 shows a simplified version of the DDG for the symptom instruction at $L18$ from Go-909. There are two objects in the DDG: The first one is reachable from a real-pointer, which means it's dependent on `malloc` ($eL1_0, eL7_0$), and the object gets marked ($eL10_0, eL12_0, eL15_0, eL18_0$). The second object is from a fake pointer; it also gets marked once in the same loop iteration as the first object ($eL10_0, eL12_0, eL15_1, eL18_1$), but has no dependencies on `malloc`.

Instead of traversing the DDG each time it needs to compute a relation, Perspect only carries out a one-pass traversal of the DDG to compute all the forward and backward relations. To compute forward relations, each node in the DDG keeps an $\mathcal{S}$-set, which is the set of all unique reachable symptom events. We initialize the $\mathcal{S}$-set of the symptom nodes to the symptom event itself. In Figure 5, $eL18_0$ and $eL18_1$'s $\mathcal{S}$-sets are initialized with themselves. Perspect then traverses the DDG in *post-order* to iteratively compute the $\mathcal{S}$-sets. For each node N, its $\mathcal{S}$-set is the union of the $\mathcal{S}$-sets of all its children nodes. (Post-order traversal guarantees that N's children are visited before N.) But keeping the $\mathcal{S}$-set of each node consumes too much memory. As an optimization, we replace

node N's $\mathcal{S}$-set with its cardinality (i.e., number of elements or $|\mathcal{S}-set|$) as soon as its $\mathcal{S}$-set is propagated to all of N's parent nodes. For a forward relation $R \triangleright (\mathcal{S}|P) = \{n_i\}$, each n_i is simply the $|\mathcal{S}-set|$ of each event of P. For example, in Figure 5, $R \triangleright (L18|L10) = \{2\}$, where 2 is the $|\mathcal{S}-set|$ of $eL10_0$; and $R \triangleright (L18|L15) = \{1,1\}$, where the two 1s come from the $|\mathcal{S}-set|$ of $eL18_0$ and $eL18_1$.

To compute backward relations, Perspect keeps a hashmap H for each symptom event. Each H keeps the number of reachable predecessor events for every corresponding predecessor instruction. For example, in Figure 5, Perspect keeps two Hs, one for $eL18_0$ and one for $eL18_1$. The H for $eL18_0$ contains 5 entries: $\{L15, L12, L10, L7, L1\}$, and the count for each key is 1. H for $eL18_1$ only contains 3 entries: $\{L15, L12, L10\}$, where the count for each key is also 1. A backward relation $R \triangleleft (L1|L18)$ is simply the set containing the count kept for $L1$ in each H, which is $\{1,0\}$.

Optimization: two-phase analysis. As an optimization, we perform our analysis in two phases. The first phase, or the "sketch" phase, only performs the analysis on the call graph. Specifically, each node in the SDG in this phase is a function, and each edge is a function invocation. The exceptions are functions that contain the symptom instructions: we directly connect the symptom instructions to the entry of these functions. We do not perform the expensive data-flow analysis in the sketch phase. Given this SDG, we build relations using the same algorithm: first obtain the DDG from the sketch SDG, and perform the relation analysis on this DDG. So, the P in the relations $R_?(P,\mathcal{S})$ we obtained is a function. For P whose relation changes, we zoom into P and perform the full data- and control-flow analysis described in §4.1. This optimization allows us to avoid the expensive dependency computations on functions that are not relevant to the root cause; it is particularly effective in large code bases like MongoDB where the symptom often has a deep call stack. In practice, this optimization reduces Perspect's static analysis time by 10 times.

4.3 Handling Binary Difference

Perspect is able to compare relations generated from different binaries by matching each binary instruction to its corresponding one in the other binary, or between different binaries generated from the same source code (i.e., compiled for the 64- and 32-bit machines). Perspect first performs the source-level `diff` to establish the file and line number mapping between two versions. However, a line in the source code often compiles to multiple binary instructions, sometimes even multiple basic blocks of binary instructions. So we cannot only rely on source-level line number mapping to map binary instructions. Instead, for two binary instructions to be considered as the same between two version, they have to have 1) the mapping source-level line number, and 2) the same binary basic block number, assigned according to the postorder traversal of all the basic blocks of the same source code line, and 3) the same offset within the basic block. If the instruction is not found at the same offset, Perspect also searches for nearby instructions.

5 Experimental Evaluation

Perspect's premise is that relational debugging can automatically and effectively locate the root causes of real-world performance problems that are hard to diagnose by existing tools. We validate these hypotheses with three evaluation questions: 1) Can Perspect effectively locate the root cause of challenging performance problems? 2) Can Perspect's output, in the form of relations, help users understand root causes? 3) What is the analysis time of Perspect?

- §5.1: Perspect effectively locates root causes of evaluated performance problems in Go runtime, MongoDB, Redis, and Coreutils. Perspect also correctly excludes a root cause from application code when it is in the OS kernel.
- §5.2: The output of Perspect, in the form of relations, can speed up debugging time by at least 10.87 times.
- §5.3: Perspect diagnoses 10/12 of the issues in 8 minutes on average, and diagnoses the other two in a few hours.

Target applications and performance problems. We evaluate Perspect on twelve real-world performance issues of four applications: the Go runtime, MongoDB, Redis, and Coreutils. All three are complex software systems, consisting of more than 220K, 6,955K, 37K, and 456K lines of code, respectively. The performance problems are collected from the issue trackers of the target applications, based on keywords like "performance", "slow", "degrade", etc. Where possible, we focus on high-priority issues that cannot be simply answered by using a profiler but take significant human time and effort, as those are the problems that need advanced tools like Perspect.

We then try to reproduce these issues based on the steps described in the issue reports. Reproducing performance problems is nontrivial and time-consuming—many of the issues are imprecisely described (e.g., no version information or reproduction steps) and are hard to reproduce. In total, it took several person-months for us to prepare the dataset. We realize that our dataset has several "famous" bugs (e.g., Go-909 in §2) because they have more detailed information for reproduction.

As shown in Table 2, the twelve issues cover different symptoms and use cases. In terms of symptoms, nine caused slowdown; three caused memory overuse, including bloated heap size and resident set size (the amount of memory used by the process). There are three different types of performance baselines: five are from a different version, one from different hardware architecture, and the other five are from different inputs. Notably, we evaluated *two open issues* where developers were unable to diagnose them (MongoDB-56274 and -57221).

Inputs. Perspect takes as inputs of the reproduction of the performance problems. We directly used reproduction programs

	Issue	Description	Metric	Succ?	Rank	Abs. pred.?	Cand. relns.	SDG size	DDG size
Go runtime	909	Fake pointers stops GC from freeing dead objects	heap	Yes	1st	No	1	16054	516k
	7330	Performance of operator += is worse than single +	time	Yes	1st	Yes	1	26	200k
	8832	Hugepage promotion causes memory bloat	RSS	Partial	-	-	-	3140	6851
	11068	Printing is very slow for large Floats	time	Yes	1st	Yes	1	10650	1409k
	12228	More aggressive GC degrades performance	time	Partial	-	Yes	-	9886	39745
	13552	Not recycling large stack spans leaks memory	RSS	Yes	1st	No	1	18060	55580
Mongo	44991	Erroneous cache clear for common prefixed keys	time	Yes	1st	Yes	1	2109	461k
	56274	Slow when deleting opposite to search order	time	Yes	1st	No	3	56	6132
	57221	Slow due to moving cursor of obsolete query plan	time	Yes	1st	No	3	5100	268k
Redis 7595		performance downgrade after enabling TLS	time	Yes	1st	Yes	1	35	801
Core	930965	seq 84x slower with –equal-width	time	Yes	1st	Yes	1	668	20002
	1014738	du –exclude 4x slower when given a trivial string	time	Yes	1st	Yes	1	7563	20784

Table 2: **Perspect's result on 12 real-world performance issues across 4 systems: Go runtime, MongoDB ("Mongo"), Redis, and Coreutils ("Core").** Mongo-56274 and -57221 are two *open bugs*. "Metric" shows the type of performance metric that describes the symptoms. "Succ?" shows whether Perspect *successfully* locates the root cause. "Rank" shows the ranking of root-cause relations. "Abs. pred?" tells whether the root-cause relations break any absolute predicates. "Cand. relns." shows the number of root-cause candidate relations. Where Perspect returns a pair of forward and backward relations, it is counted as one root cause candidate. "SDG size" and "DDG size" show the average SDG and DDG size from the good and bad runs, in terms of the number of instructions and their runtime instances, respectively.

attached in the reports, or created reproductions by closely following the descriptions in the reports. We find that except for Go-909, which provided three similar reproductions, all issues describe at most one good and one bad execution. Perspect is able to exploit high repetitiveness of runtime events within one execution, and works with two executions as is.

5.1 Effectiveness

Table 2 shows the effectiveness of Perspect in diagnosing the twelve performance bugs. The overall results are very positive. Perspect successfully locates the root causes for ten performance problems, and ranks the root-cause relation as the highest (or the only) suspect. Eight of them are closed issues and we use the criteria that the reported root cause has to be captured by the output relations of Perspect. For the two open bugs, the relations output by Perspect provided explanations of the root causes that were confirmed by the developers.

Perspect partially locates the root causes of the other two issues (Go-8832 and Go-12228). For Go-8832, Perspect correctly excludes the root cause (which lies in Linux) from the Go runtime. For Go-12228, the source codes changed significantly; Perspect is unable to map the relations across the executions. In this case, Perspect outputs the relations between the symptoms and causal predecessors so that a human developer can complete the rest of the debugging process.

As shown in Table 2, Perspect is able to effectively nail down a very small set of root-cause candidate relations. This is attributed to its iterative filtering (§3.3.2) and refinement (§3.3.3); Our experiments confirm that the relations between most events and their direct successors do not change across executions. Perspect also filters out most causally related events with low contributions to the symptoms.

Note that 10/12 of the evaluated issues have no clear-cut failures—they are reported because the programs ran slower or consumed more memory than their respective baselines; the remaining two only *occasionally* result in out-of-memory errors (Go-909 and Go-13552). Hence, those issues can hardly be diagnosed by tools for functional failures. In at least four issues, the root causes do not manifest in any absolute predicate changes—the relations captured by Perspect show that the root causes exist in both executions, only their distributions differ. Lastly, as shown by the sizes of SDGs and DDGs, there are too many causally related instructions and runtime events—causality analysis alone can hardly pinpoint the root cause in code.

We discussed how Perspect locates the root causes of Go-909 and MongoDB-57221 in §2. We briefly present a few more.

Mongodb-44991. Mongodb-44991 is major performance regression introduced in v4.2.1 and took developers several days to diagnose. Figure 6 shows the simplified code containing the root cause. As a memory optimization, Mongodb stores key prefixes only once per page [5]; hence, it needs to decompress a key before evicting it back to disk. If the same key has been decompressed before, Mongodb copies the cached data directly (L4) to avoid building the key from scratch (L6). In v4.2.1, L11 was erroneously added, which clears the `size` variable, effectively invalidating cached data (L4).

Perspect takes the inputs of two executions from the good version (v4.0.13) and the buggy version (v4.2.1) as reported

```
1 void convert(void *key) {...
2   get_key_info(key, &data, &size);
3   if (... && size > 0)
4     memcopy(key, data, size); // fast path
5   else
6     build_key(entry, key); // slow path
7 }
8 void get_key_info(void *key, void *data, int *size) {
9   data = get_data(key);
10  ...
11  size = 0;
12 }
```

Invalidated the condition of using cached data (fast path); Erroneously introduced in v4.2.1.

Figure 6: The root cause of MongoDB-44991 (used in §5.2).

in the issue. Perspect locates a pair of relations in the bad run: $R\blacktriangleleft\blacktriangleright(L6, L11)$ and reports it as the highest-ranked root cause.

Go-13552. Developers noticed that the RSS slowly creeped over to 1GB, even though the actual heap usage stayed below 4MB [6]. Diagnosing this bug took 5 days, and the developers eliminated several wrong guesses before nailing down the root cause. First, they had a hard time deciding whether the problem came from the heap or the stack. Once they focused on the stack, they further thought that the memory bloat was due to normal stack spans not being recycled fast enough. Finally, they found the root cause to be a special type of large stack spans which were not recycled at all.

Perspect ranks a relation $R\blacktriangleright$(`sysMmap|allocLarge`) the highest, indicating that the increased mmap allocations are for large-sized stack spans. This connects the two essential pieces of information together to pinpoint the bug.

MongoDB-56274 (open issue). MongoDB-56274 is another open issue we diagnosed using Perspect, and the root cause has been confirmed by developers. The developers noticed that deleting records in descending order was twice as slow as in ascending order. MongoDB deletes records iteratively: after it deletes the record, it searches for the next record to delete. The `search` function has a hard-coded order: it always looks for the next record in ascending order first (`search_forward`); if no record is found, it searches backwards in descending order (`search_backward`). Hence, when the deletion order is the same as the search order, the next record is always found immediately; but, when the deletion order is the opposite, MongoDB traverses through many deleted records, then searches in the opposite direction, causing the slowdown.

Perspect locates the root cause to the hard-coded search order logic; In particular, it identifies three relations that increased significantly in the bad run: 1) $R\blacktriangleright$(`search_backward|search`): in the good run, `search_backward` is rarely invoked, as the next record is always immediately located by `search_forward`; 2) $R\blacktriangleright$(`prev_record|search_backward`) and 3) $R\blacktriangleright$(`next_record|search_forward`) indicates increased number of records traversed in both directions of search.

Go-8832. Developers observed unexpected memory bloat and mistakenly thought it was caused by bugs from Go's GC code. In fact, the root cause was Linux's promotion of huge pages in the background, which bloated the resident set size (RSS) since the distribution of the base 4KB pages was sparse. The developers spent a lot of time examining incorrect hypotheses about bugs in the GC logic, making it one of the most discussed Go performance issues.

While the current implementation of Perspect cannot analyze the OS kernel, it can help rule out wrongly suspected buggy behaviors of the Go runtime. Specifically, after comparing relations associated with the symptoms `mmap` and `munmap`, Perspect outputs no root cause candidate relations.

5.2 Usability

We evaluated the usability of Perspect with a controlled user study. We tested on 20 programmers (who are not co-author of this paper) who indicated extensive experience in debugging and GDB.

We used Go-909 and MongoDB-44991 in the study to represent resource issues and slowdowns. Each participant was asked to debug one case without any help and a different case with Perspect; so each bug has two controlled groups for comparison. For each participant, we first described the bugs and helped reproduce them. We chose one of the two cases randomly and asked the participant to diagnose it without Perspect; then for the second case, we introduced relational debugging and allowed them to use Perspect. We limited the debugging session to two hours for each bug (not including setup or reproduction time). If the time was exceeded, we considered the bug unsolved.

For Go-909, we considered a participant to have caught the root cause if they concluded that unreachable objects got marked and prevented reclamations. For MongoDB-44991, we used the criteria that the participant had to locate the instruction that clears the `size` variable eroneously (L11 in Figure 6).

Our results show that when using Perspect, participants concluded the root cause *at least 10.87 times faster* than when not using Perspect. With the help of Perspect, all participants successfully located the root causes of both issues, with an average of 10 minutes; much of the time was spent on navigating code and understanding instructions pointed to by the relations. Without Perspect, only 5/10 of the participants concluded the root causes within two hours, with an average of one hour and 47 minutes.

Interestingly, we observed that without Perspect many participants had *manual practices* like relational debugging: they printed out counters to compare occurrences of functions or instructions in the good and bad runs, and ruled out ones that did not change. However, we observed that such manual effort was neither rigorous nor systematic. For example, for Go-909, many participants examined if GC happened less often, but did not realize objects reclaimed per GC cycle changed.

We interviewed participants after the debugging session. The most overwhelming feedback is that the relation semantic is intuitive and easy to understand. One suggestion is to visualize SDGs and DDGs alongside the changed relations, which we consider implementing via GUI support.

5.3 Analysis Time

For the twelve performance issues, Perspect takes an average of under one hour to output the results. For 10/12 issues, Perspect finishes under 20 minutes, with an average of 8 minutes. The other two take an average of 5.3 hours. Most of the analysis time was spent on static and dynamic causality analysis (§3.2); the relational debugger (§3.3) takes a small fraction of the total analysis time.

Static causality analysis takes 24 minutes on average. It is bottlenecked by repeatedly invoking RR to build non-local dataflow dependencies (§4). The worst-case complexity of static causality analysis is $O(n*m)$, where n is the number of dynamic instructions executed during each RR run, and m is the number of static instructions that are causally related to the symptom instructions. The two-phase optimization described in §4.2 reduces m significantly. We can further speed up static analysis by adding more servers to parallelize the invocation of RR runs (§4.1).

Dynamic causality analysis is bottlenecked by running the instrumented program in PIN. It takes on average 35 minutes across the 12 issues (<20 minutes for 10/12 issues). Perspect effectively reduces the DDGs's sizes by sampling one symptom event out of N, while keeping a large number of symptom events to maintain statistical significance.

In comparison, the relational debugger only takes a small fraction of the total dynamic analysis time, typically a few minutes. Reducing the size of the DDG also effectively reduces the average complexity of the relational debugger, which has a worst-case complexity of $O(p^2)$, where p is the number of instructions executed that are causally relevant to the symptom events.

6 Discussion and Limitations

Relational debugging provides a new way of understanding performance problems. We find it generally applicable to many challenging performance problems that do no manifest via clear-cut predicates. Relational debugging assumes that the relations in the executions are statistically significant. It is possible that an execution is too short. On the other hand, our evaluation shows that the executions based on the reproduction steps documented in real-world issue reports are mostly sufficient—there are enough repetitive patterns for Perspect to be effective. It is straightforward to apply Perspect to multiple runs if one is too short.

Our current implementation of Perspect shares some limitations of its building blocks. Specifically, Perspect cannot debug performance problems that are non-deterministic (e.g., they depend on the scheduling and timing of events), because Perspect uses deterministic replay (RR [7]) and its dynamic instrumentation could change the timing. Please note: this does *not* mean that Perspect cannot debug multi-threaded systems—all the evaluated systems (except Coreutils) are multi-threaded. In fact, it is reported that the vast majority (>90%) of real-world performance problems are deterministic [28].

Perspect currently only supports native code. We plan to implement relational debugging for applications in managed languages like Java. We believe the implementation can be built on the JVM Tool Interface. Perspect can be easily extended to handle additional language constructs like exception handling etc. [4] We will also explore how to apply relational debugging to performance problems of distributed systems by analyzing relations of distributed events. Perspect can be extended to support metrics such as P95 latency etc. [5]

7 Related Work

Performance debugging with Perspect takes three steps: 1) identifying symptoms, 2) causality analysis, and 3) relational debugging for automatically pinpointing root causes. We discuss related work based on the three components.

Automatic performance debugging/diagnosis. The closest related work (in terms of locating root causes in code) is [39], which applies statistical debugging [32] to performance problems. The essential idea of statistical debugging is to identify predicates that have strong correlations with the failure. However, as we have shown in this paper, it is fundamentally limited to performance problems that manifest via absolute predicates. Moreover, since statistical debugging in [39] does not take causality into consideration, many of the observed predicates could be irrelevant to the symptom; To compensate, it requires a large number of highly variable good and bad executions. Another related work is X-ray [15] which summarizes performance costs of runtime events and attributes them to input and configuration values w.r.t the symptom. Different from Perspect, X-ray is designed for end users (e.g., sysadmins) and does not target root causes in the code. X-ray uses differential performance summarization which identifies branches where execution paths diverge and reasons about the performance difference between the two branch outcomes. In this sense, it also focuses on divergence of predicates between executions.

There are tools for debugging special types of performance problems with predefined patterns, such as loops [35, 40, 44],

[4] When Perspect detects a symptom instruction is causally related to an exception handler, it can perform the analysis at instructions that can potentially throw an exception that is caught by this handler, treating these instructions as symptom instructions.

[5] Instead of calculating weighted sums, Perspect can perform the z-test on the weight of each symptom event against the distribution of weights of all symptom events (symptom events with a z-test score of 1.645 corresponds to the 95th percentile). The rank of each relation can be the number of causally related outlier symptom events.

memory leaks [41], and data locality [30]. Differently, Perspect is designed to be a general debugging/diagnosis tool.

Automatic functional failure debugging/diagnosis. Prior studies developed techniques to pinpoint the root causes of functional failures in code, based on invariant analysis [24, 26, 38], log analysis [50] and statistical debugging [32]. Perspect focuses primarily on performance problems which have very different characteristics from function failures.

Causality analysis. Perspect applies relational debugging to instructions and their runtime events that are causally related to the symptoms. Many advanced techniques have been developed for causality analysis [16, 22, 29, 34, 37, 42, 45–47, 49, 53]. Perspect can potentially use them to enhance its causality analysis (§3.2). For example, we can further accelerate the causality analysis, learning from failure sketching [29], REPT [22] and ER [53] that use Intel PT to efficiently trace causally dependent instructions and augment the trace with symbolic execution [18, 21]. Argus [42] developed a way to annotate causality graphs with strong and weak edges, which can prioritize relational analysis of Perspect. SherLog [45], lprof [52], and Pensieve [49] show that runtime logs can be used with static analysis to guide the reconstruction of causal paths.

Our work is complementary to causality analysis for distributed systems (many targeting performance problems [14, 17, 20, 34, 43, 51]). Relational debugging for distributed systems based on distributed causality is our future work (§6).

Profilers. Profilers [9, 10, 12, 13, 19, 23, 36] are important utilities for performance debugging. Advanced profilers like [23, 36] can effectively identify true bottlenecks. They provide effective inputs for Perspect to locate root causes.

8 Conclusion

Debugging performance problems is (still) among the most challenging, time-consuming tasks. We presented relational debugging as a new way of understanding performance problems and locating their root causes in the code. Our key insight is that the root causes of performance bugs can be generalized to changes in relations between fine-grained runtime events, and by using relations, we capture root causes of performance bug existing semantics (such as invariants or predicates etc.) fail to capture. We developed Perspect to automate relational debugging. Perspect takes a minimal of just two executions (a good and bad run), and pinpoints the root causes of complex real-world bugs to a small number of root cause relations using an effective "filter-and-refine" algorithm. We further demonstrate Perspect's effectiveness by diagnosing two open issues which developers were unable to diagnose using existing tools. Finally, we deploy a number of carefully designed optimizations to scale Perspect to large-scale code-bases. We open-sourced Perspect and will continue improving it towards a common toolkit for performance debugging.

Acknowledgement

We thank our shepherd, Jason Flinn, and the anonymous reviewers for their feedback and comments on our work. We also thank Serguei Makarov for the suggestion to output binary instead of plain-text PIN logs for optimized performance. This work was supported by the Canada Research Chair fund, an NSERC Discovery grant, an NSERC Alliance Mission grant, and an NSF grant CNS-2130560.

References

[1] Go-1091: runtime: gob leaks memory for larger objects (above MMAP_THRESHHOLD?). https://github.com/golang/go/issues/1091, Sept. 2010.

[2] Go-909: runtime: garbage collection ineffective on 32-bit. https://github.com/golang/go/issues/909, July 2010.

[3] memory leak on 8g. https://github.com/golang/go/issues/1210, Oct. 2010.

[4] Go: Severe memory problems on 32bit Linux. https://news.ycombinator.com/item?id=3805302, 2012.

[5] File formats and compression . http://source.wiredtiger.com/2.3.0/file_formats.html, 2014.

[6] Go-13552: runtime: RSS creeps over 1GB even though heap is 4MB. https://github.com/golang/go/issues/13552, 2015.

[7] rr: lightweight recording and deterministic debugging. https://rr-project.org/, 2017.

[8] Paradyn/Dyninst - Welcome | Putting the Performance in High Performance Computing. https://www.dyninst.org/, 2021.

[9] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2021.

[10] SystemTap. https://sourceware.org/systemtap/, 2021.

[11] Pin: A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2022.

[12] The GNU Profiler. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html, 2022.

[13] Valgrind: a memory profiling and debugging tool. https://valgrind.org/, 2022.

[14] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Oct. 2003).

[15] ATTARIYAN, M., CHOW, M., AND FLINN, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).

[16] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Oct. 2010).

[17] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[18] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (Dec. 2008).

[19] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC'04)* (June 2004).

[20] CHEN, A., WU, Y., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)* (Aug. 2016).

[21] CHIPOUNOV, V., KUZNETSOV, V., AND CANDEA, G. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)* (Mar. 2011).

[22] CUI, W., GE, X., KASIKCI, B., NIU, B., SHARMA, U., WANG, R., AND YUN, I. REPT: Reverse Debugging of Failures in Deployed Software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[23] CURTSINGER, C., AND BERGER, E. D. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[24] ERNST, M. D., COCKRELL, J., GRISWOLD, W. G., AND NOTKIN, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)* (May 1999).

[25] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349.

[26] HANGAL, S., AND LAM, M. S. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)* (May 2002).

[27] HODGES, J. J. The significance probability of the smirnov two-sample test. *Arkiv fiur Matematik*, 3 (1958), 469–486.

[28] JIN, G., SONG, L., SHI, X., SCHERPELZ, J., AND LU, S. Understanding and Detecting Real-World Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)* (June 2012).

[29] KASIKCI, B., SCHUBERT, B., PEREIRA, C., POKAM, G., AND CANDEA, G. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[30] KHAN, T. A., NEAL, I., POKAM, G., MOZAFARI, B., AND KASIKCI, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)* (July 2021).

[31] LI, Y., TAN, T., MØLLER, A., AND SMARAGDAKIS, Y. Scalability-First Pointer Analysis with Self-Tuning Context-Sensitivity. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)* (Nov. 2018).

[32] LIBLIT, B., NAIK, M., ZHENG, A. X., AIKEN, A., AND JORDAN, M. I. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (June 2005).

[33] LINDEN, G. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, Nov. 2017.

[34] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[35] NISTOR, A., SONG, L., MARINOV, D., AND LU, S. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)* (May 2013).

[36] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)* (May 2015).

[37] RAVINDRANATH, L., PADHYE, J., AGARWAL, S., MAHAJAN, R., OBERMILLER, I., AND SHAYANDEH, S. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).

[38] SAHOO, S. K., CRISWELL, J., GEIGLE, C., AND ADVE, V. Using Likely Invariants for Automated Software Fault Localization. In *Proceedings of the 18th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)* (Mar. 2013).

[39] SONG, L., AND LU, S. Statistical Debugging for Real-World Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'14)* (Oct. 2014).

[40] SONG, L., AND LU, S. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)* (May 2017).

[41] VILK, J., AND BERGER, E. D. BLeak: Automatically Debugging Memory Leaks in Web Applications. In *Proceedings of*

the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18) (June 2018).

[42] WENG, L., HUANG, P., NIEH, J., AND YANG, J. Argus: Debugging Performance Issues in Modern Desktop Applications with Annotated Causal Tracing. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)* (July 2021).

[43] WU, Y., ZHAO, M., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. Diagnosing Missing Events in Distributed Systems with Negative Provenance. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)* (Oct. 2014).

[44] XIAO, X., HAN, S., ZHANG, D., AND XIE, T. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13)* (July 2013).

[45] YUAN, D., MAI, H., XIONG, W., TAN, L., ZHOU, Y., AND PASUPATHY, S. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-XV)* (March 2010).

[46] ZAMFIR, C., AND CANDEA, G. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys'10)* (Apr. 2012).

[47] ZAMFIR, C., KASIKCI, B., KINDER, J., BUGNION, E., AND CANDEA, G. Automated Debugging for Arbitrarily Long Executions. In *Proceedings of the 14th Workshop on Operating Systems (HotOS-XIV)* (May 2013).

[48] ZELLER, A., AND HILDEBRANDT, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng. 28*, 2 (Feb. 2002), 183–200.

[49] ZHANG, Y., MAKAROV, S., REN, X., LION, D., AND YUAN, D. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[50] ZHANG, Y., RODRIGUES, K., LUO, Y., STUMM, M., AND YUAN, D. The Inflection Point Hypothesis: A Principled Debugging Approach for Locating the Root Cause of a Failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (Oct. 2019).

[51] ZHAO, X., RODRIGUES, K., LUO, Y., YUAN, D., AND STUMM, M. Non-intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *Proceedings of the 12th Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[52] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. Lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[53] ZUO, G., MA, J., QUINN, A., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)* (June 2021).