Relational Debugging — Pinpointing Root Causes of Performance Problems

Xiang (Jenny) Ren, Sitao Wang, Zhuqi Jin, David Lion, Adrian Chiu, Tianyin Xu, Ding Yuan
Performance issues are costly

“Google found a 0.5 seconds delay (in page load time) caused a 20% decrease in repeat traffic”

“the Go process has been crashing every other hour ... it was such a memory hog”
Performance is **relative**

Request rate changed?
- More requests/period

Request type changed?
- More allocation/request

Memory leak?
- Fewer deallocation/allocation

?
Performance is *relative*

Idea: locate most specific reference point to captures the root cause
Performance is relative

Idea: locate most specific reference point to captures the root cause
Existing solutions are limited

Statistical debugging
• Identifies **absolute** predicates correlated with failure
• Requires labeling **many** executions as fail or success

```
flag = true;
... 
if (flag) {
    ...
    throw(ex);
}
```

```
flag = false;
... 
if (flag) {
    }
```

Failure: throw(ex)  Predicate: flag == true  Correlation: 100%
Relational Debugging
– pinpoints root causes of performance problems

**Relations** between events represents relative performance & general representation of performance root causes.

\[ R(\text{malloc}|\text{request}) = \{20, 40\} \]
Relational Debugging – pinpoints root causes of performance problems

Relations between events represents relative performance & general representation of performance root causes.

Request#1
Mem usage:
200MB

Request#2
Mem usage:
400MB

\( R(\text{malloc(size)}|\text{request}) \)

= \{200MB, 400MB\}

Relations can represent:
- Memory usage
- CPU cycles
- Network bandwidth
- Disk usage
...
Relational Debugging – pinpoints root causes of performance problems

**Relations** between events represents relative performance & general representation of performance root causes.

\[ R(\text{malloc}|\text{request}) = 30 \]
Relational Debugging – pinpoints root causes of performance problems

Relations between events represents relative performance & general representation of performance root causes.

\[ R(B|A) = \mu \]

“The # of event B’s that causally dependent on an event A.”
Relational Debugging
– pinpoints root causes of performance problems

Challenges
• Possibles relations in an execution are combinatorial
• Which ones capture the root cause of performance bug?

./program

main() {
    ...
    if (flag) {
        ...
    } else {
        ...
    }
}
Relational Debugging
– pinpoints root causes of performance problems

Core idea:
locate most specific reference point to capture the root cause

main() {
    while (true) {
        handle_request();
    }
}
Relational Debugging
– pinpoints root causes of performance problems

Core idea:
locate most specific reference point to capture the root cause

```c
main() {
    while (true) {
        handle_request();
    }
}
```

Given $R(\text{handle\_request} | \text{main}()) = 10$ ➔ $10$

$R(\text{malloc} | \text{main}()) = 2\text{GB} ➔ 6\text{GB}$
Relational Debugging
– pinpoints root causes of performance problems

Core idea:
locate most specific reference point to capture the root cause

```c
main() {
    while (true) {
        handle_request();
    }
}
```

Given

\[ R(\text{handle\_request}|\text{main}()) = 10 \Rightarrow 10 \]

Refine to

\[ R(\text{malloc}|\text{handle\_request}) = 205\text{MB} \Rightarrow 315\text{MB} \]

\[ R(\text{malloc}|\text{main}()) = 2\text{GB} \Rightarrow 6\text{GB} \]
Perspect implements Relational Debugging

Root cause candidates

$R (\ldots | \ldots)$ impact 99% rank #1
$R (\ldots | \ldots)$ impact 50% rank #2

...
**Perspect** implements Relational Debugging

Causal analysis
- Bootstrap with performance symptoms
- Identify causal predecessors of the symptoms

Relational debugging
**Step1. Build** relations at most general reference points
**Step2. Filter** relations that have not changed
**Step3. Refine** relations - move ref. points closer to symptom
**Step4. Rank** root cause candidates based on impact on perf.

Repeat
Go-909 – A memory leak bug

Go-909 causes “Severe memory problems on 32bit Linux”

```go
for i := 0; i < 1000; i++ {
  r := make([]float64, 923521)
}
```

- Impacted many workloads & Extensively discussed
Diagnosing Go-909 was challenging

- Diagnosed through trial-and-error after more than a year
- Root cause breaks no program invariants/absolute predicates
The root cause of Go-909

```c
void *p = malloc(...);
const int q = 0x8126890;
```

![Diagram](live_dead_objects.png)
The root cause of Go-909

```c
void *p = malloc(...);
const int q = 0x8126890;
```

- **Live object**
  - Mark object as reachable
  - GC: mark

- **Dead object**
  - GC: reclaim
The root cause of Go-909

```c
void *p = malloc(...);
const int q = 0x8126890;
```

![Live object]

```
GC: mark object as reachable
```

![Dead object]

```
GC: mark object as reachable
```

The root cause of Go-909
Perspect pinpoints the root cause of Go-909

void *p = malloc(...);  
const int q = 0x8126890;

Live object: GC: mark object as reachable  
Dead object: GC: mark object as reachable

Root cause relation:
Good run: $R(\text{malloc}|\text{mark\_object}) = \{1, 1, 1, 1, \ldots 1, 0\}$
Bad run: $R(\text{malloc}|\text{mark\_object}) = \{0, 0, 0, 0, \ldots 0, 1\}$

"The # of malloc events each mark event depends on."
Perspect pinpoints the root cause of Go-909

mark() { ... if points_to_heap(ptr) {mark(*ptr)} }

void *p = malloc(...); const int q = 0x8126890;

Live object

Dead object

GC: mark object as reachable

Root cause relation:
Good run: \( R(\text{malloc}|\text{mark\_object}) = \{1, 1, 1, 1, ... 1, 0\} \)
Bad run: \( R(\text{malloc}|\text{mark\_object}) = \{0, 0, 0, 0, ... 0, 1\} \)

“The # of malloc events each mark event depends on.”
Perspect pinpoints the root cause of Go-909

```
mark() { ... if points_to_heap(ptr) {mark(*ptr)} }
```

```
void *p = malloc(...);   const int q = 0x8126890;
```

**Live object**  
**Dead object**

**Root cause relation:**

Good run: \( R(\text{malloc}|\text{mark\_object}) = 0.99 \)

Bad run: \( R(\text{malloc}|\text{mark\_object}) = 0.01 \)

Impact: 99% rank: 1/1
Causal analysis

- Bootstrap with performance symptoms
- Identify causal predecessors of the symptoms

Relational debugging

Step 1. Build relations
Step 2. Filter relations
Step 3. Refine relations
Step 4. Rank root cause candidates

Repeat
Bootstrap with performance symptoms

```c
malloc()
heap_size += obj.size;
```

```c
reclaim(obj)
heap_size -= obj.size;
```
Identify causal dependencies of the symptoms

- **malloc()**
  - heap_size += obj.size;

- **mark()**
  - if(points_to_heap(ptr))
    - mark(*ptr)

- **sweep()**
  - if(!obj.marked)
    - reclaim(obj)
    - heap_size -= obj.size;
Perspect automates relational debugging

Causal analysis
- Bootstrap with performance symptoms
- Identify causal predecessors of the symptoms

Relational debugging
Step 1. Build relations
Step 2. Filter relations
Step 3. Refine relations
Step 4. Rank root cause candidates

Repeat
Step 1. Build relations at most general reference points

```
heap_size += obj.size;

if (points_to_heap(ptr))
    mark(*ptr)

if (!obj.marked)
    reclaim(obj)
heap_size -= obj.size;
```
Step1. Build relations at most general reference points

malloc()

mark()

if(points_to_heap(ptr))

mark(*ptr)

if(!obj.marked)

reclaim(obj)

heap_size -= obj.size;
Step1. Build relations at most general reference points

```
heap_size -= obj.size;

reclaim(obj)

if(!obj.marked)
    if(points_to_heap(ptr))
        mark(*ptr)

malloc()
mark()
sweep()
```

heap_size -= obj.size;
reclaim(obj)
Step 1. Build relations at most general reference points

\[ \text{heap}_\text{size} \text{ += obj.size;} \]

\[ \text{malloc}() \]

\[ \text{mark}() \]

\[ \text{if(points_to_heap(ptr))} \]

\[ \text{mark}(*\text{ptr}) \]

\[ \text{mark}() \]

\[ \text{sweep}() \]

\[ \text{if(!obj.marked)} \]

\[ \text{reclaim(obj)} \]

\[ \text{heap}_\text{size} \text{ -= obj.size;} \]

\( R(\text{reclaim}|\text{malloc}()) \)

\( R(\text{reclaim}|\text{mark}()) \)

\( R(\text{reclaim}|\text{sweep}()) \)
Step1. Build relations at most general reference points

malloc() → mark() → if(points_to_heap(ptr)) → mark(*ptr) → reclaim(obj)

heap_size += obj.size;

malloc() → mark() → sweep() → if(!obj.marked) → reclaim(obj)

heap_size -= obj.size;

R(reclaim|malloc())  R(reclaim|mark())  R(reclaim|sweep())
Refine relations – rule #1

```
sweep()
for (obj in heap_span)
  if (!obj.marked)
    reclaim(obj)
```

\( R(\text{reclaim}|\text{sweep}()) \)
Refine relations – rule #1

```
sweep()

for (obj in heap_span)
  if (!obj.marked)
    reclaim(obj)

R(reclaim|sweep())
```

Refine

```
R(reclaim|if(!obj.marked))
```
Refine relations – rule #1

- `sweep()`
  - `for (obj in heap_span)`
    - `if(!obj.marked)`
      - `reclaim(obj)`

Refine relations:

- `R(reclaim|sweep())`
  - `R(reclaim|if(!obj.marked))`

OK since

- `R(if(!obj.marked)|sweep())`
Step2. Refine relations to capture root cause

```cpp
malloc()
    |   mark()
    |  if(points_to_heap(ptr))
    |      mark(*ptr)
    |        sweep()
    |            if(!obj.marked)
    |               reclaim(obj)
    |                 heap_size -= obj.size;
```

R(reclaim|sweep())
Step2. Refine relations to capture root cause

- `malloc()`
- `mark()`
  - `if(points_to_heap(ptr))`
  - `mark(*ptr)`
- `sweep()`
  - `if(!obj.marked)`
  - `reclaim(obj)`
    - `heap_size -= obj.size;`
  - `R(reclaim|sweep())`

ROOT CAUSE
- `R(malloc|mark)`
Evaluating *Perspect*'s effectiveness

- Evaluated on 12 bugs from Golang, Mongodb, Redis, Coreutils:
  - 10 bugs: *Perspect* diagnosed the root cause *successfully*
  - 1 bug: root cause in kernel, excluded from go system
  - 1 bug: unsuccessful due to significant code change

- Diagnosed two *open* bugs

“*[Perspect’s result] ties all the pieces together into a nice explanation.*”

—MongoDB developer’s comment
Perspect's usability and scalability

• Participants diagnose 2 cases 10.87 X faster with Perspect: Go-909 and Mongodb-44991

• Perspect takes an average of 8 minutes to run on most cases
Related work

Statistical debugging
• Identifies absolute predicates correlated with failure
• Requires labeling many executions as fail or success

X-Ray
• Captures root causes in input parameters & configurations

Other solutions
• Designed for specific patterns of bad performance
Relational Debugging

- *Relation* btw. events captures relativeness of performance bugs
- Refine relations to narrow down to most specific root causes

*Perspect* (implements relational debugging)

- Pinpoints root causes of complex real-world bugs efficiently
- Helped diagnose two *open bugs*

https://gitlab.dsrg.utoronto.ca/dsrg/perspect