Exploring the Past

A typical debugging session looks like this:

1. Set a breakpoint
2. Start program, reaching breakpoint
3. Step, Step, Step, …
4. Oops! I’ve gone too far!
How does it work?

- ODB records a trace of the entire execution history
- Slows down programs by a factor of 10
- Records about 100 MB/s
- Now available in commercial tools

Dynamic Slicing

- Static slices apply to all program runs:
  - General + reusable, but imprecise
- A dynamic slice applies to a single run:
  - Specific and precise
**Static Slicing**

- Given a statement $B$, the backward slice contains all statements that could influence the read variables or execution of $B$.
- Formally: $S^B(B) = \{A | A \rightarrow^* B\}$

---

1. Obtain a trace of the execution
2. Get the variables that are read and written
3. Assign an empty slice to each written variable
4. Compute the slices from start to end:

$\text{DynSlice}(w) = \bigcup_i (\text{DynSlice}(r_i) \cup \{\text{line}(r_i)\})$
```plaintext
n = read();
a = read();
x = 1;
b = a + x;
a = a + 1;
i = 1;
s = 0;
while (i <= n) {
    if (b > 0)
        if (a > 1)
            s = s + x;
i = i + 1;
}
write(s);
```

DynSlice(w) = \( \bigcup_i (\text{DynSlice}(r_i) \cup \{\text{line}(r_i)\}) \)

Trace | Write | Read | Dynamic Slice
--- | --- | --- | ---
1 | n | a | 2
2 | a | a | 2
3 | x = 1; | x | 2
4 | b = a + x; | b, a, x | 2, 3
5 | a = a + 1; | a | 2
6 | i = 1; | i | 2
7 | s = 0; | s | 2
8 | while (i <= n) { | p8 | i, n | 6, 1
9 | if (b > 0) | p9 | b, p8 | 4, 2, 3, 8, 6, 1
10 | if (a > 1) | p10 | a, p9 | 5, 2, 9, 4, 2, 3, 8, 6, 1
12 | s = s + x; | s, x, p8 | 7, 3, 8, 6, 1
13 | i = i + 1; | i, p8 | 8, 6, 1
8 | while (i <= n) { | p8 | i, n | 13, 8, 6, 1
9 | if (b > 0) | p9 | b, p8 | 4, 2, 3, 13, 8, 6, 1
10 | if (a > 1) | p10 | a, p9 | 5, 2, 9, 4, 2, 3, 13, 8, 6, 1
12 | s = s + x; | s, x, p8 | 12, 7, 3, 6, 8, 1, 13
13 | i = i + 1; | i, p8 | 13, 8, 6, 1
8 | while (i <= n) { | p8 | i, n | 13, 8, 6, 1
15 | write(s); | o15 | s | 12, 7, 3, 6, 8, 1, 13
```
\begin{verbatim}
1 n = read();
2 a = read();
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
\end{verbatim}

Static slice for (s, 15)  Dynamic slice for (s, 15)

\section*{Discussion}

- Dynamic slices are much more precise than static slices (applied to the one run, that is)
- From some variable, a backward slice encompasses on average
- 30% of the entire program (static slice)
- 5% of the executed program (dynamic slice)
- Overhead as in omniscient debugging

Ko and Myers (2004) from CMU (Human–Computer Interaction)
“Why did” questions

- Take the dynamic slice of the variable
- Follow at most two dependencies
- If programmer wants to, follow dependencies transitively

Ko and Myers (2004) from CMU (Human–Computer Interaction)

Ko and Myers (2004) from CMU (Human–Computer Interaction)
[switch back and forth between last slide and this slide]
“Why did $s = 2$ in Line 15?”

```plaintext
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```

“Why didn’t $x = 2$ in Line 11?”

```plaintext
1 n = read(); // n = 2
2 a = read(); // a = 0
3 x = 1;
4 b = a + x;
5 a = a + 1;
6 i = 1;
7 s = 0;
8 while (i <= n) {
9     if (b > 0)
10         if (a > 1)
11             x = 2;
12     s = s + x;
13     i = i + 1;
14 }
15 write(s);
```
Discussion

The WHYLINE combines

• omniscient debugging
• static slicing
• dynamic slicing

in an attractive package, showcasing the state of the art in interactive debugging

Tracking Infections

1. Start with the infected value as seen in the failure
2. Follow back the dependencies
3. Observe and judge origins – are they sane?
4. If some origin is infected, repeat at Step 2
5. All origins are sane? Here’s the infection site!

Concepts

★ Omniscient debugging allows for simple exploration of the entire execution history
★ Dynamic slicing tells the origin of a value
★ To track down an infection, follow dependencies and observe origins, repeating the process for infected origins