Visualizing Program State In The Pernosco Debugger

Robert O'Callahan*

Work done in collaboration with Kyle Huey.

* Currently at Google Deepmind. The work described in this talk predates my Google role.

Overview

- Introduction to omniscient debugging
- The design principles of Pernosco
- Some novel visualizations of program state
- What we learned
- Reflecting on the “debuginfo problem”

Omniscient Debugging

What is debugging?

Examining program states to understand why something (bad) happens.

These days, mostly logging.

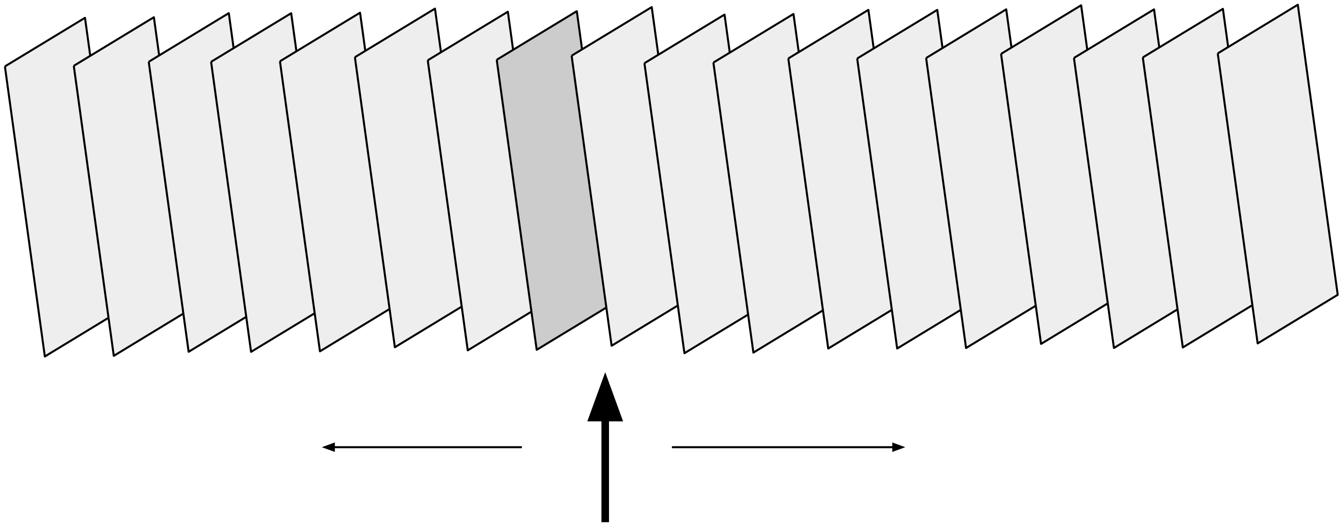
Interactive debuggers

Software tools that

- Monitor program execution
- Stop at desired points
- Report information about the program state at those points

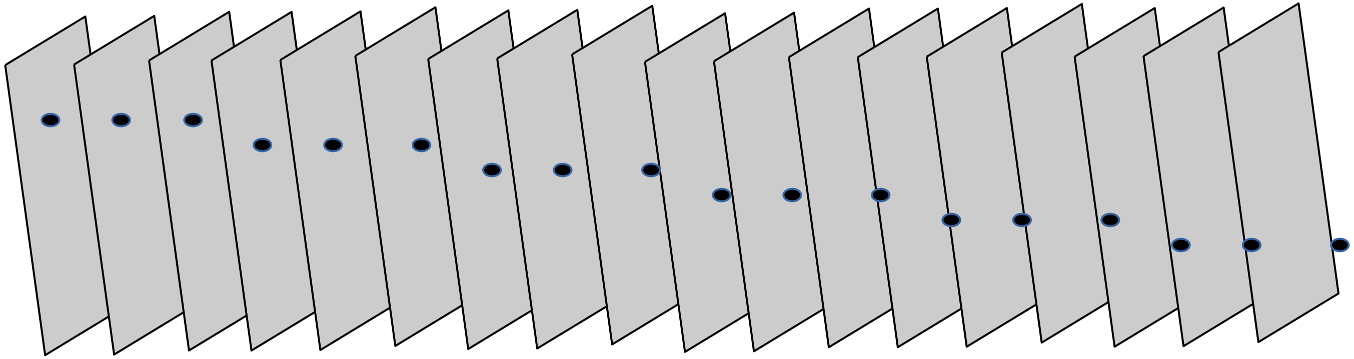
Single Program State

Traditional debuggers (e.g. GDB) only show a single program state at a chosen time



Omniscience

What if we had instant access to all program states?



Omniscience

Conceptually: put all program states into a database and query it — “Omniscient Debugging” (Bil Lewis, ODB)

Rethink the debugger interface from the ground up:

Given fast access to all program states, what user interface lets developers fix bugs the fastest?

Can we implement it in practice?

Pernosco

I wanted to work on these questions.
2016: With Kyle Huey, started a company to build and sell omniscient debugging (bootstrapped).

Raising another question: *Will people pay for it?*

We built it!

Pernosco workflow

- Users create **recordings** of program execution using rr (a “record and replay” debugger)
- Submit recordings to cloud or on-prem server
- Server builds a compressed database of machine-level states by *replaying with binary instrumentation*
- Users access Web-based debugger UI
- Debugger UI *queries* data via RPC

Affordable, timely omniscience

All memory and register states; efficient queries for:

- Value of register/memory range at any time
- Time of last write to register/memory range
- Every time instruction executed at address

Other data, e.g. function calls

Clever compression: < 1 byte per instruction executed

Build DB for 0.5T instructions in less than an hour

Practicality

Works for big applications: Firefox, Chrome, JVM

A *very modestly* successful business.

- So take what I say with caution.
- But some users really love the product.

Design Principles

Visualize state across time

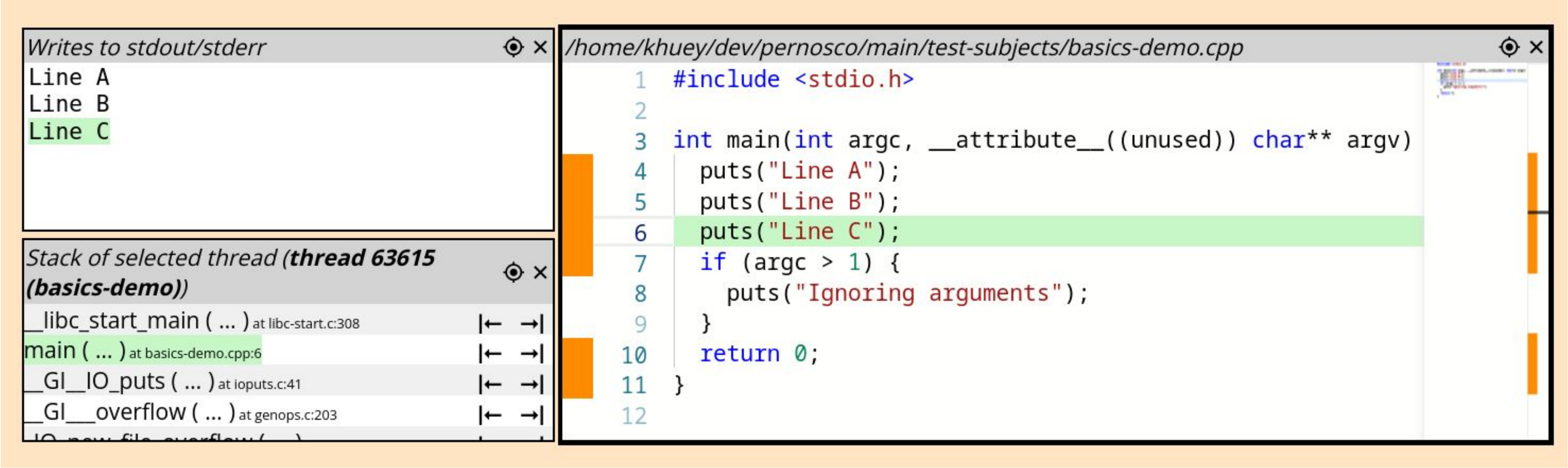
Previously users must build up a picture of events across time manually

- Single-stepping or stopping at breakpoints

→ Present events across time in a single visualization

No stepping!!!

Classic UI, reinterpreted

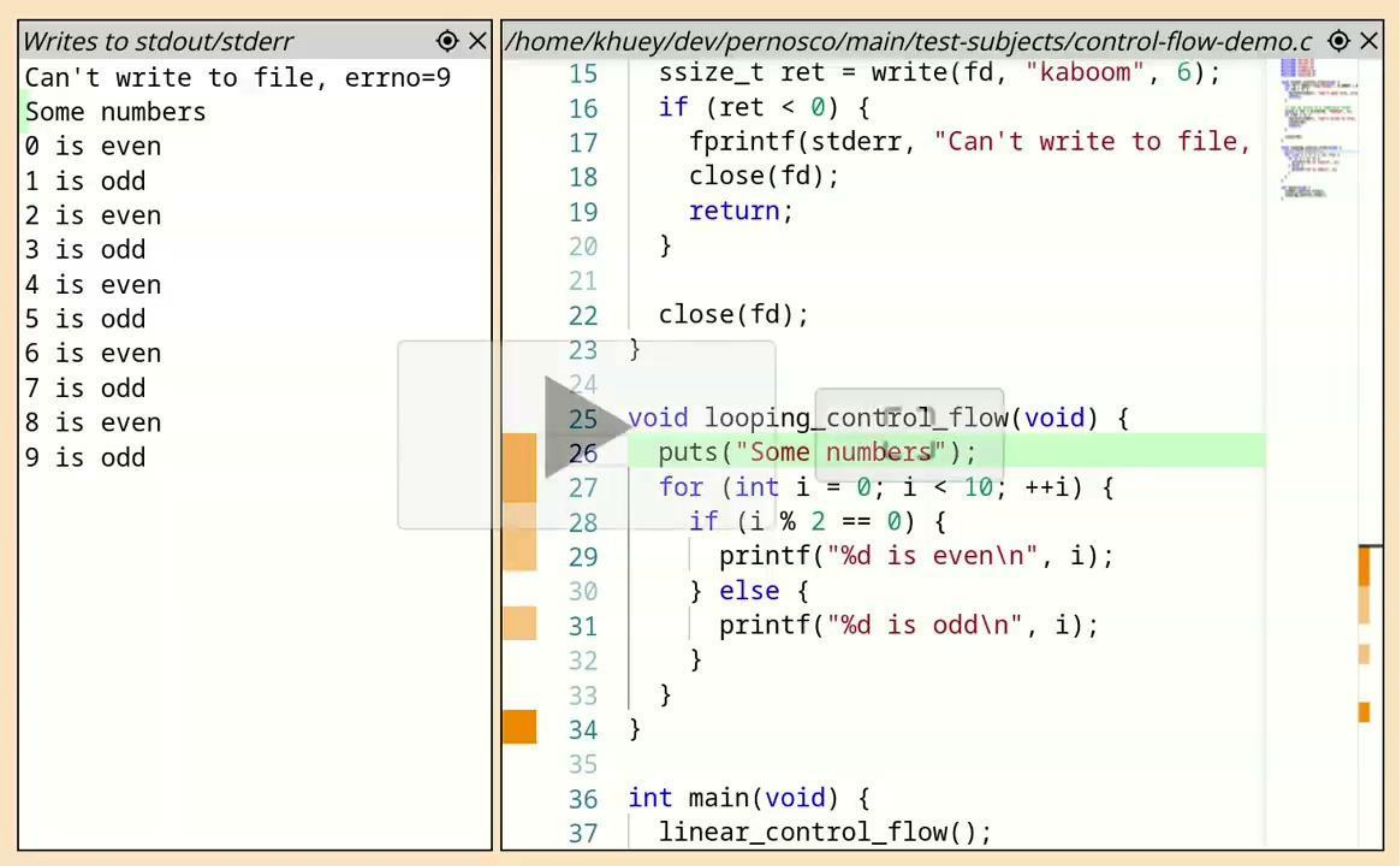


Query panes

Each pane visualizes the results of a query that produces a sequence of events

A “focus” (current point in time) cursor is present in each pane

Click to focus on an event


Writes to stdout/stderr
Can't write to file, errno=9
Some numbers
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
/home/khuey/dev/pernosco/main/test-subjects/control-flow-demo.c
ssize_t ret = write(fd, "kaboom", 6);
if (ret < 0) {
fprintf(stderr, "Can't write to file,
close(fd);
return;
}
close(fd);
}
void looping_control_flow(void) {
puts("Some numbers");
for (int i = 0; i < 10; ++i) {
if (i % 2 == 0) {
printf("%d is even\n", i);
} else {
printf("%d is odd\n", i);
}
}
}
int main(void) {
linear_control_flow();

Stick to UX “critical paths”

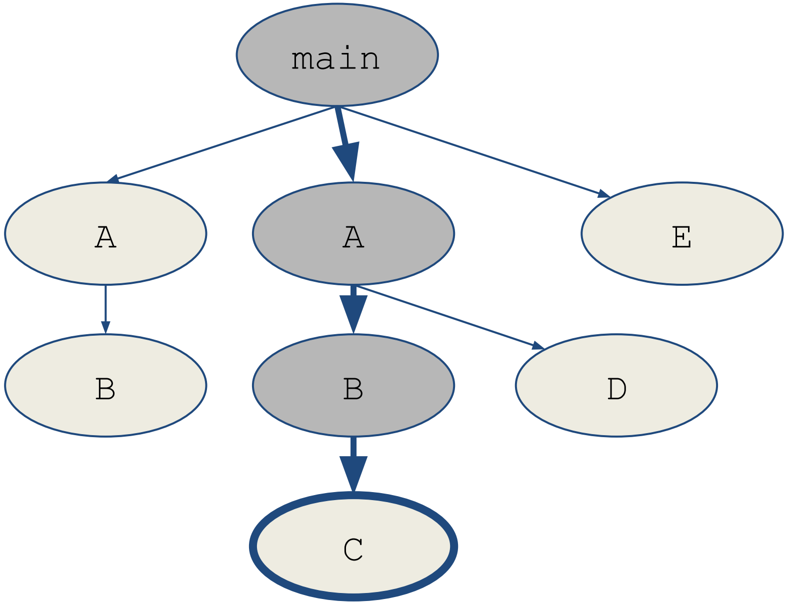
Only add a feature if there exists a specific use-case where the **best** user experience requires that feature.

- A feature might be cool or sometimes useful; that’s insufficient.

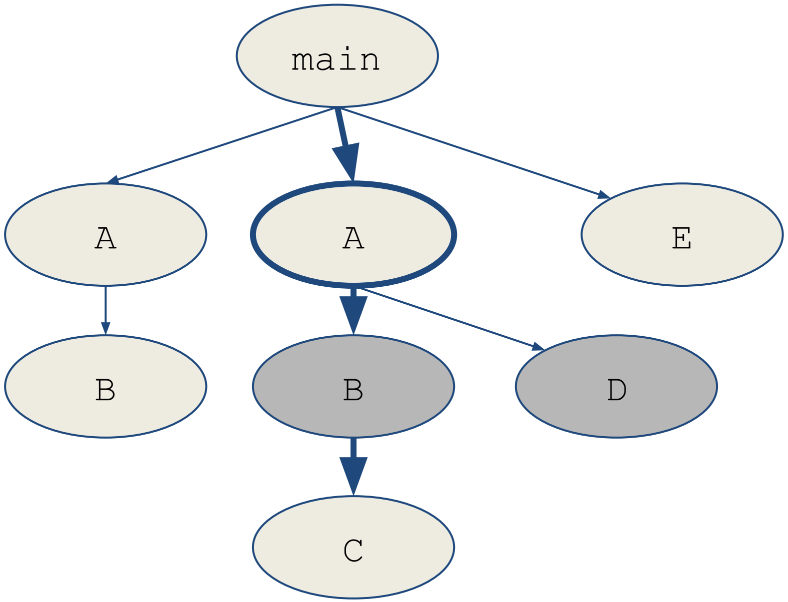
Example: no video-like scrubber bar!

Generalize existing abstractions

Example: from call stacks to **call trees**.



Example: from call stacks to **call trees**.



Callees view

```
static already_AddRefed<nsFontMetrics> GetInflatedFontMetricsForFrame(
    const nsIFrame* aFrame) {
  return GetFontMetricsForFrame(aFrame, FontSizeInflationFor(aFrame));
}
```

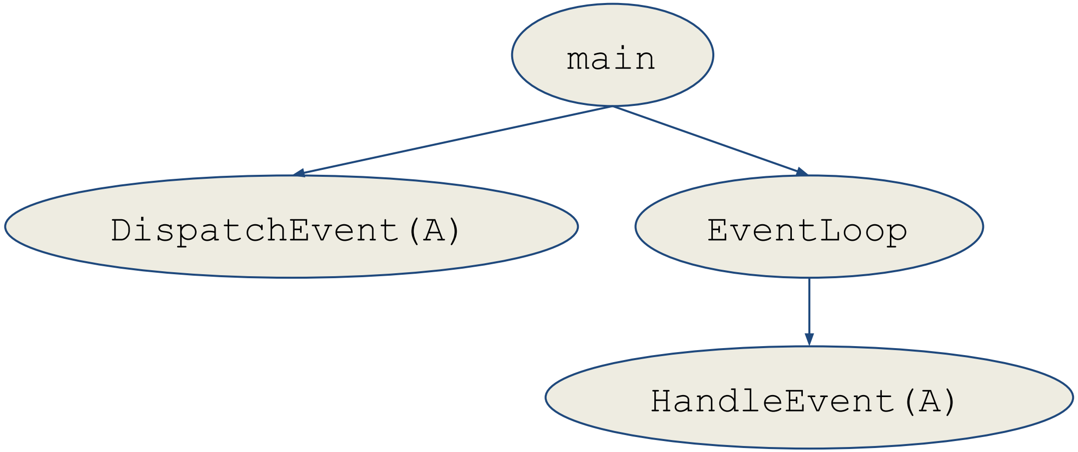
Callees of nsLayoutUtils::GetInflatedFontMetricsForFrame

nsLayoutUtils::FontSizeInflationFor (aFrame=0x7f0f054b1a80 ▼) = 1

nsLayoutUtils::GetFontMetricsForFrame (aFrame=0x7f0f054b1a80 ▼ , aInflation=1) = {mRawPtr=0x7f0f09668b00 ▼ }

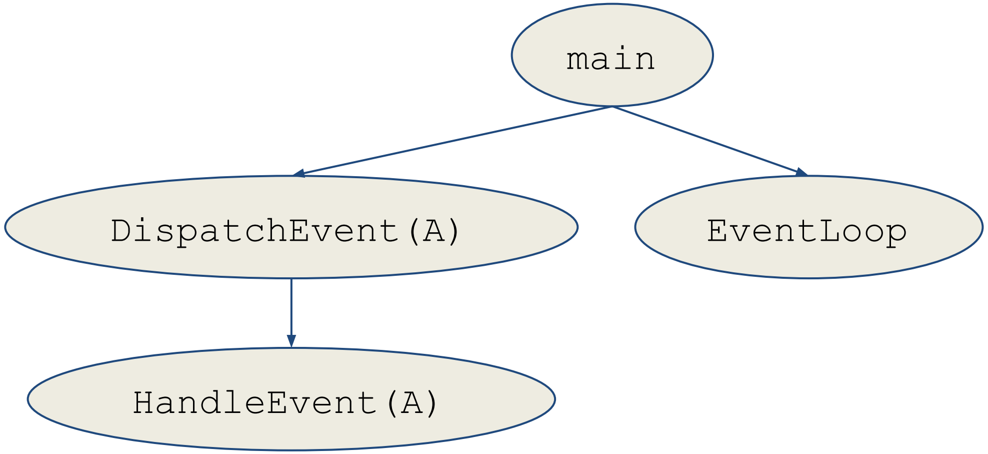
Generalizing further

Higher-level control trees:



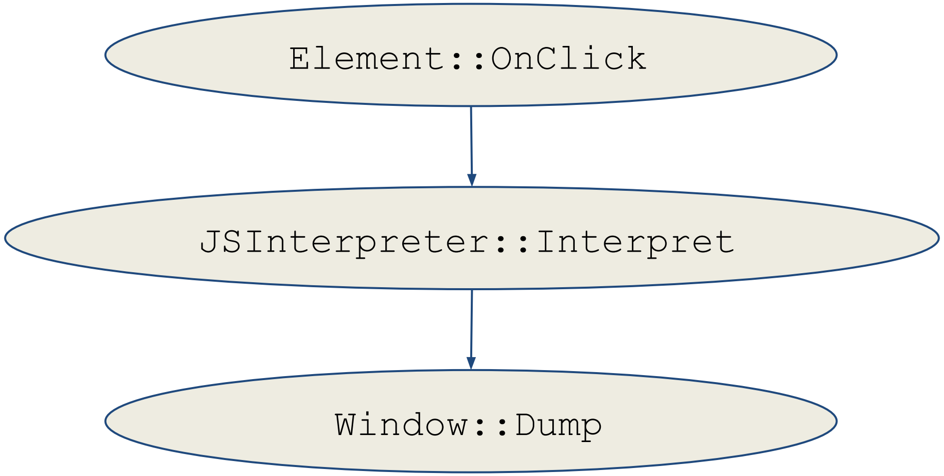
Generalizing further

Higher-level control trees:



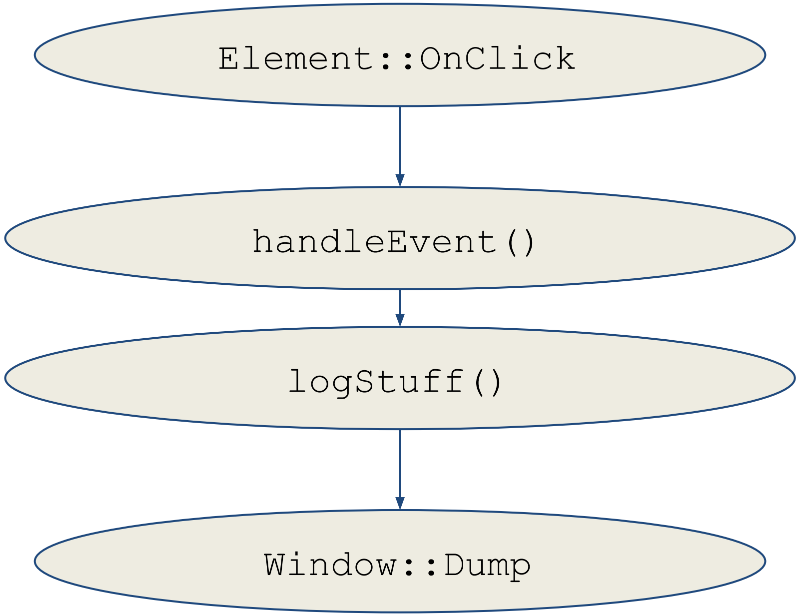
Generalizing further

Even higher level:



Generalizing further

Even higher level:



Instant responses

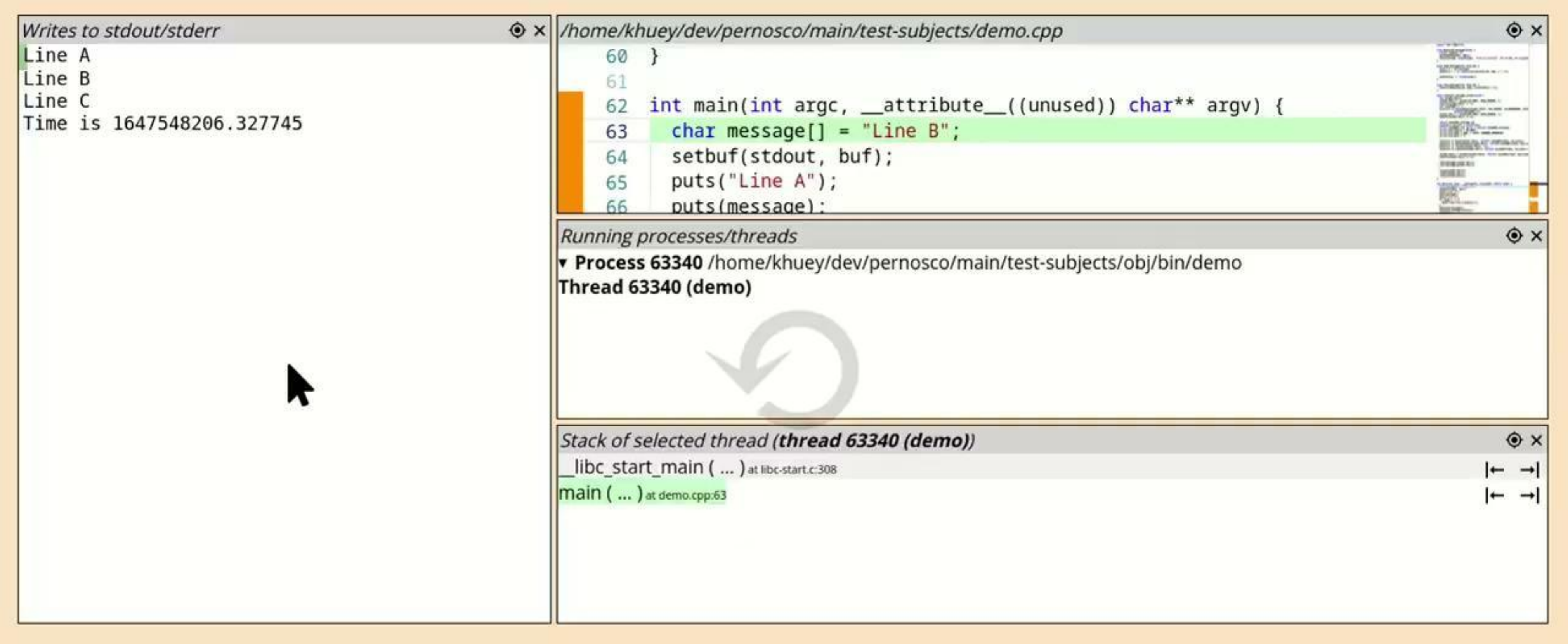
Obvious?

Omniscience → No re-execution, just database queries

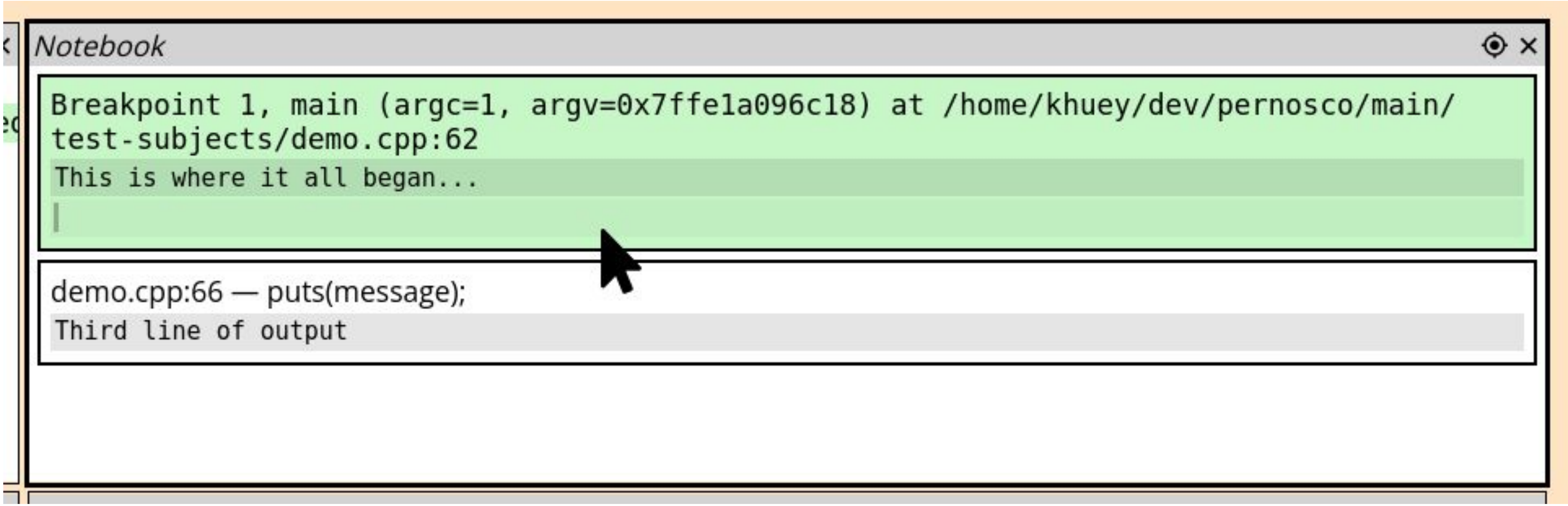
“Performance is the best feature”

Step change → stay in flow

Direct answers: Dataflow

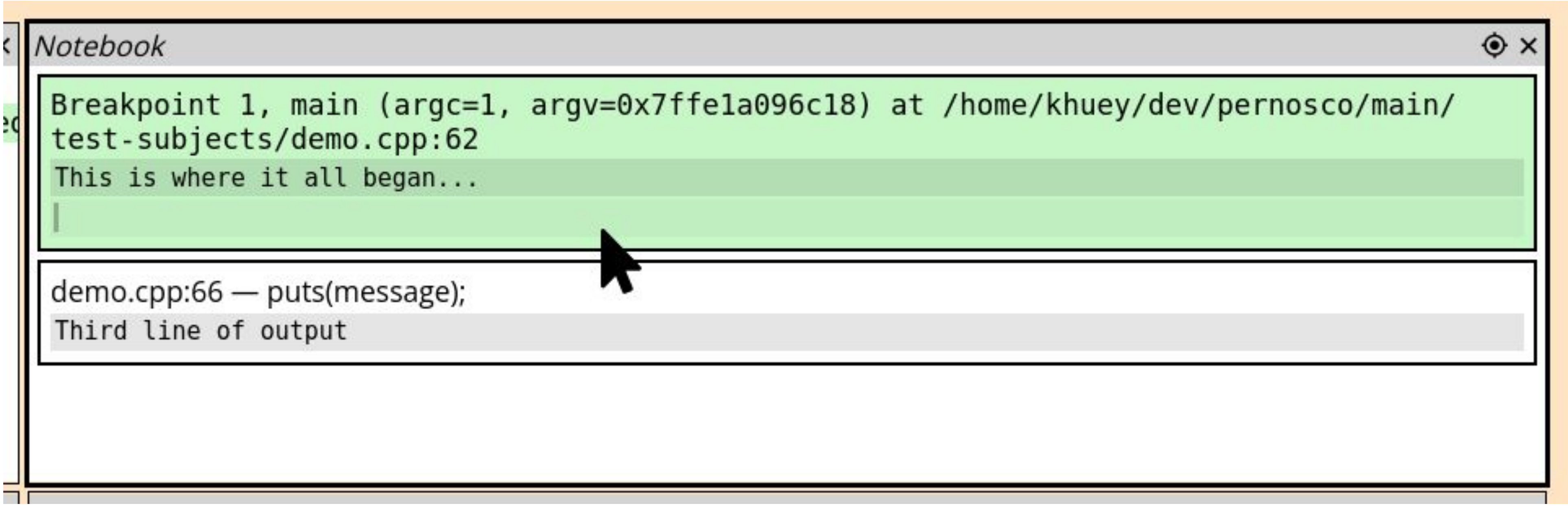


Support user habits:
Debugging notebook



Thin Web client, server state

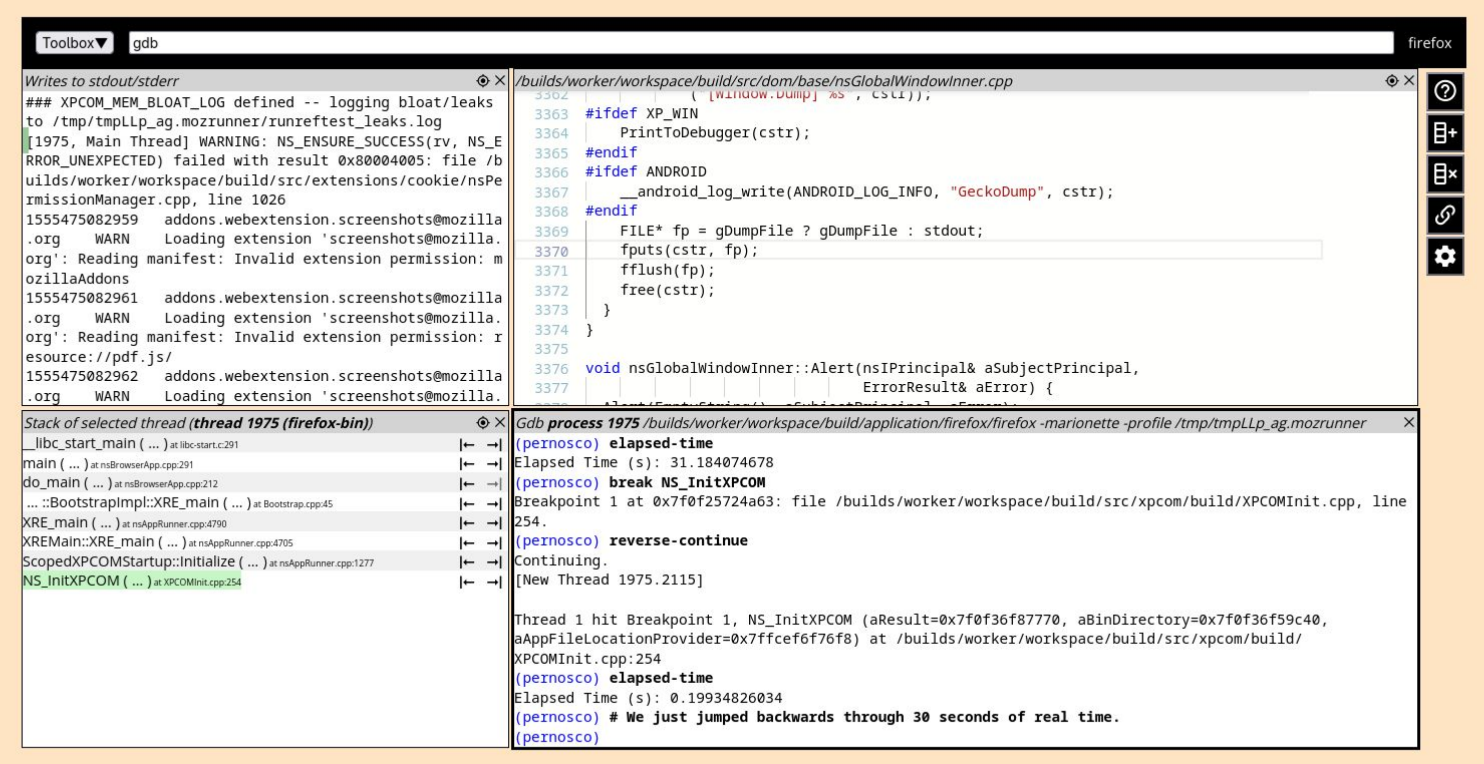
→Easy collaboration



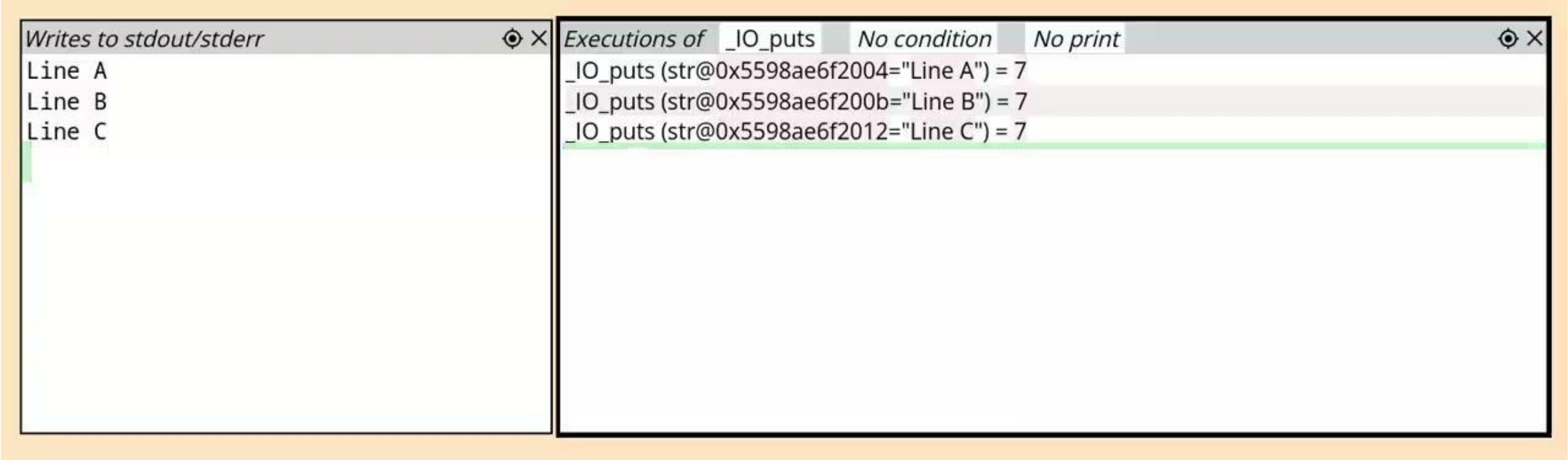
Unify paradigms

“Breakpoints and logging are kind of the same”

Unify paradigms



Composability



What We Learned

Pernosco usability

Some users get it right away without help!!!

- This suggests we got something right.

Other users bounce off

- Less familiar than traditional debuggers

More to learn here

Pernosco Adoption

✅ Some customers!

✅ Profitable subscription model!

✅ Customers love the product

🙁 Selling debuggers is incredibly hard

Bets worked out

Omniscient engine scales surprisingly well.

Stateless thin client works well in practice.

All kinds of debugging features fit into our framework.

Some challenges:

- “User function calls” create counterfactual worlds, e.g. allocating memory for a string that never was

Worked out less well

Omniscience scaling still has limits.

Hard to compete with the instant gratification of logging.

Pernosco has to be carefully integrated into users' workflows.

Some problems not yet solved

User-customizable control and data abstractions

Leveraging differences between runs for the “why didn’t this happen” problem

Understanding what happens *within a statement* can be hard — more later!

Source-level Mapping

All debuggers have to deal with this

For C/C++/Rust etc:
Compilers generate DWARF describing a mapping from machine state to source-language state:

- Line number tables map PCs to source lines (+cols?)
- Variable tables map PC+var to expression and type
  - Debugger evaluates expression to bytes, renders bytes using the type
- Unwinding tables describe how to walk stack frames

Complications

Optimizations require complex debuginfo

E.g. inlined functions have to be represented

- “PC 0xFF is in A() inlined into B()”

E.g. variables may not even exist

- “At PC 0xFF, variable X **<optimized out>**”

Debuginfo sucks

Correct and complete debuginfo is low priority for compiler vendors
→ Debuggers don’t work well
→ Users don’t use them
→ Return to step 1

Omniscience helps!

~~Unwinding tables~~

- Use call history instead

“At PC 0xFF, variable X **<optimized out>**”

- Look backwards in time for the last point where it was available!
- Requires DWARF extension…

Missing features

DWARF does not describe the return values of functions

- Get them via the ABI, but not for inlined functions

We would like to be able to display the values of all executed **(sub)expressions** (especially identifiers)

- DWARF doesn’t support this
- It would be super expensive
- It would require mapping instructions to particular source tokens accurately, which is hard

Identifier mapping

Want to know what each source identifier refers to

- E.g. for subexpressions
- E.g. for navigation
- DWARF doesn’t have it, might be expensive

On-demand debuginfo

Generating all possibly-needed information up-front may not be tenable.

Can we run the compiler lazily?

- Reproducible builds — maybe yes!
- Often a wide separation between build and debugging environments in practice

Semantic analysis

Humans can interpret machine states with source code and a little bit of help

- E.g. follow dataflow to observe which register a variable is being stored in

Can we use LLMs or some other technique?
Can we make that reliable and scalable enough to supplement or replace traditional debuginfo?
Problems…

Source code understanding

Debuggers don’t parse/understand source code.

- Just a block of text!
- Separation between build and debugging environments makes it impractical
- And languages are so complicated, so many versions…

I don’t know how to solve this in a practical way.

Conclusions

Omniscient debugging is not that hard to implement.

It enables many powerful visualizations and other features users find appealing

- Especially debugging at higher levels of abstraction
- There’s much more that can be done!

Mapping machine states to source level is far from a solved problem in practice. Need radical new solutions!