

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.

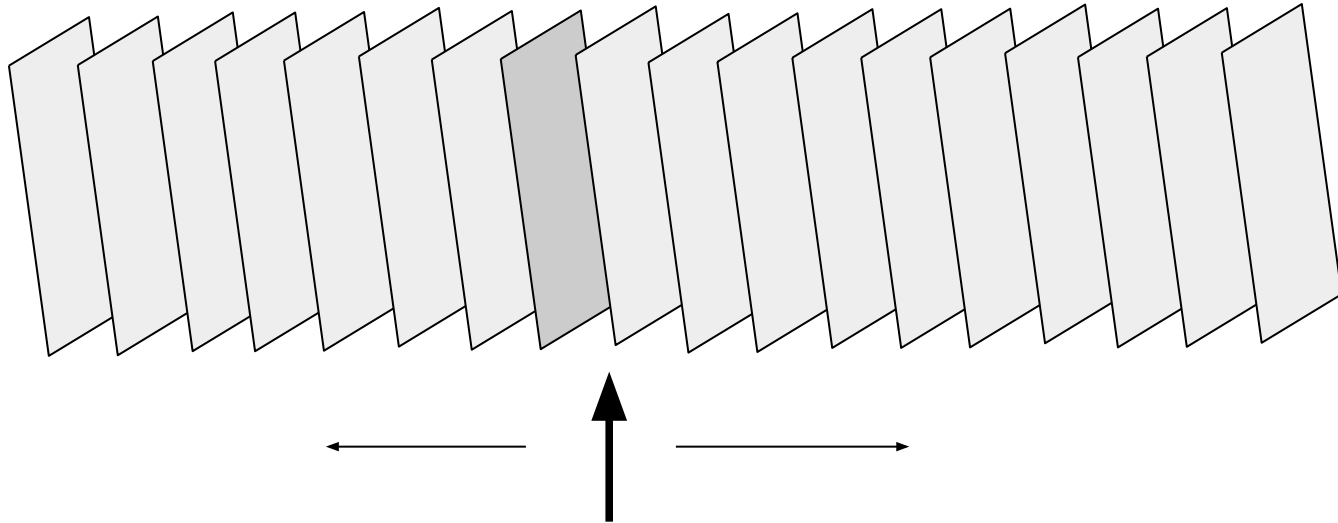
Interactive debuggers

Software tools that

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

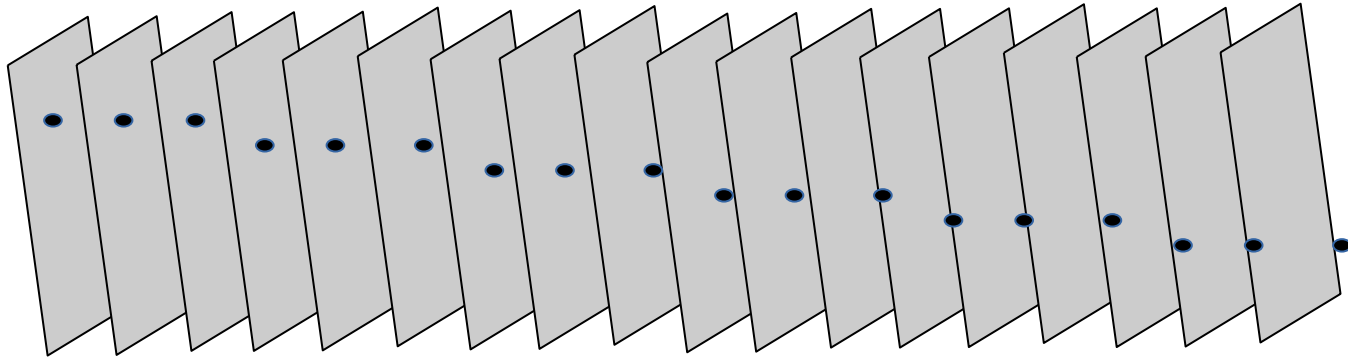
Single Program State

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

Writes to stdout/stderr

Line A
Line B
Line C

Stack of selected thread (thread 63615 (basics-demo))

__libc_start_main (...) at libc-start.c:308	← →
main (...) at basics-demo.cpp:6	← →
_GI_IO_puts (...) at ioputs.c:41	← →
_GI__overflow (...) at genops.c:203	← →
IO_new_file_overflow (...)	← →

/home/khuey/dev/pernosco/main/test-subjects/basics-demo.cpp

```
1  #include <stdio.h>
2
3  int main(int argc, __attribute__((unused)) char** argv)
4  {
5      puts("Line A");
6      puts("Line B");
7      puts("Line C");
8      if (argc > 1) {
9          puts("Ignoring arguments");
10     }
11     return 0;
12 }
```

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
15  ssize_t ret = write(fd, "kaboom", 6);
16  if (ret < 0) {
17      fprintf(stderr, "Can't write to file,
18      close(fd);
19      return;
20  }
21
22  close(fd);
23  }
24
25  void looping_control_flow(void) {
26      puts("Some numbers");
27      for (int i = 0; i < 10; ++i) {
28          if (i % 2 == 0) {
29              printf("%d is even\n", i);
30          } else {
31              printf("%d is odd\n", i);
32          }
33      }
34  }
35
36  int main(void) {
37      linear_control_flow();
```

Stick to UX “critical paths”

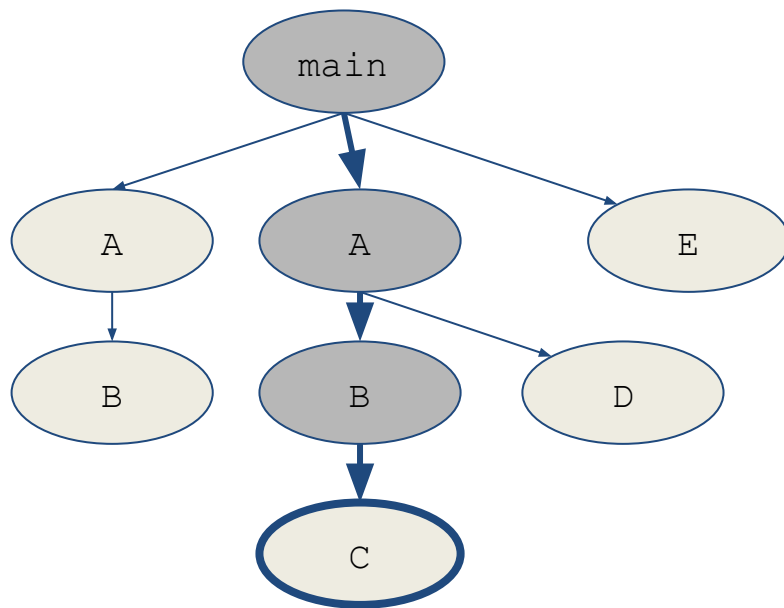
Only add a feature if there exists a specific use-case where the **best** user experience requires that feature.

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

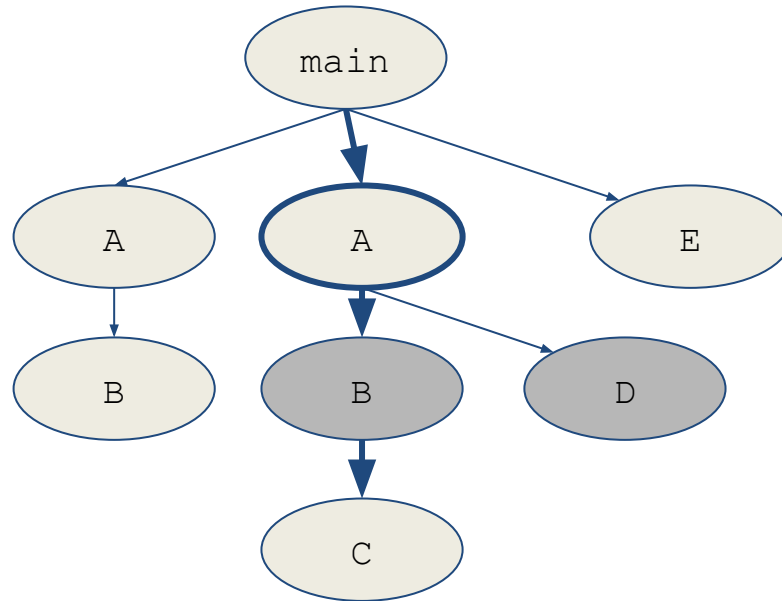
Example: no video-like scrubber bar!

Generalize existing abstractions

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



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



Callees view

```
1327  
1328     static already_AddRefed<nsFontMetrics> GetInflatedFontMetricsForFrame(  
1329         const nsIFrame* aFrame) {  
1330         return GetFontMetricsForFrame(aFrame, FontSizeInflationFor(aFrame));  
1331     }  
1332
```

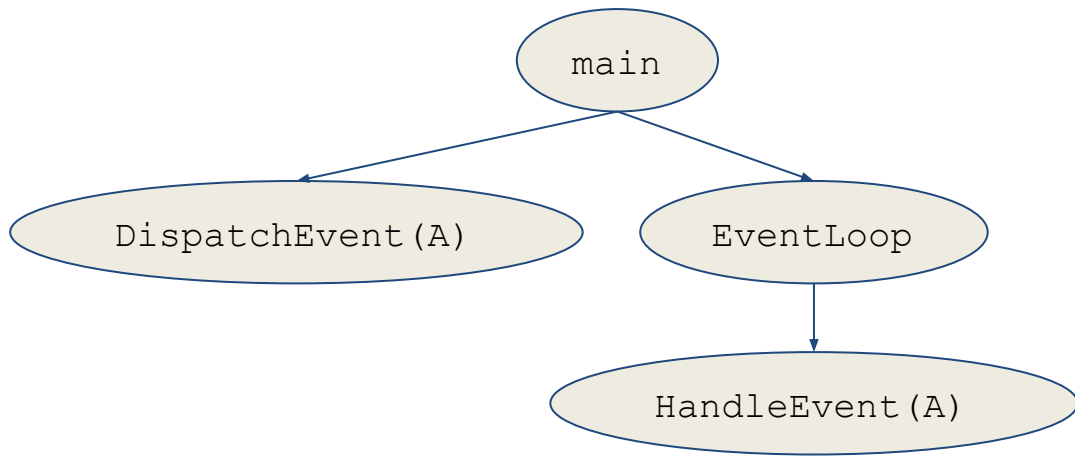
Callees of nsLayoutUtils::GetInflatedFontMetricsForFrame  

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

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

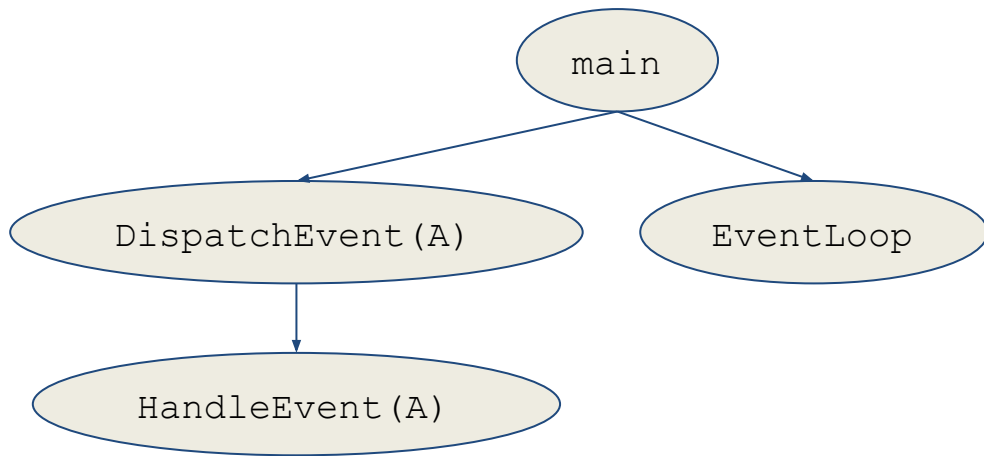
Generalizing further

Higher-level control trees:



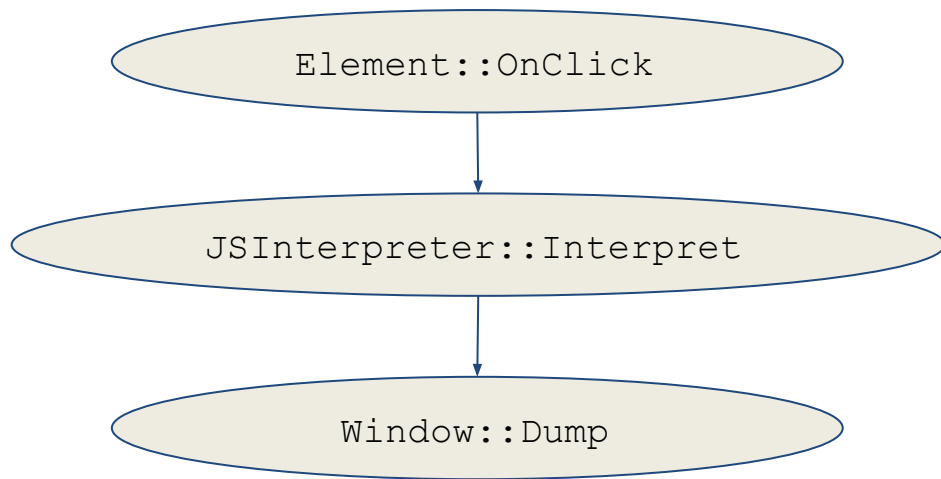
Generalizing further

Higher-level control trees:



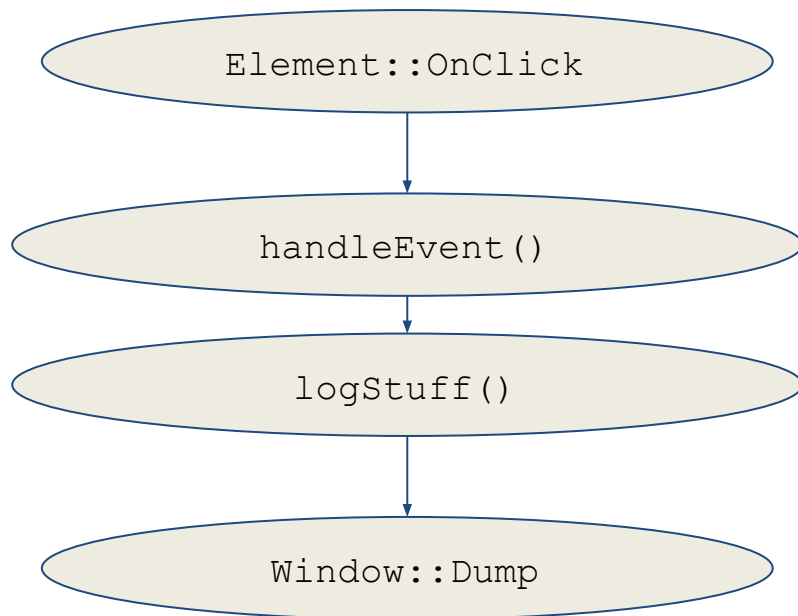
Generalizing further

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

Writes to stdout/stderr

Line A
Line B
Line C
Time is 1647548206.327745

/home/khuey/dev/pernosco/main/test-subjects/demo.cpp

```
60 }  
61  
62 int main(int argc, __attribute__((unused)) char** argv) {  
63     char message[] = "Line B";  
64     setbuf(stdout, buf);  
65     puts("Line A");  
66     puts(message);  
}
```

Running processes/threads

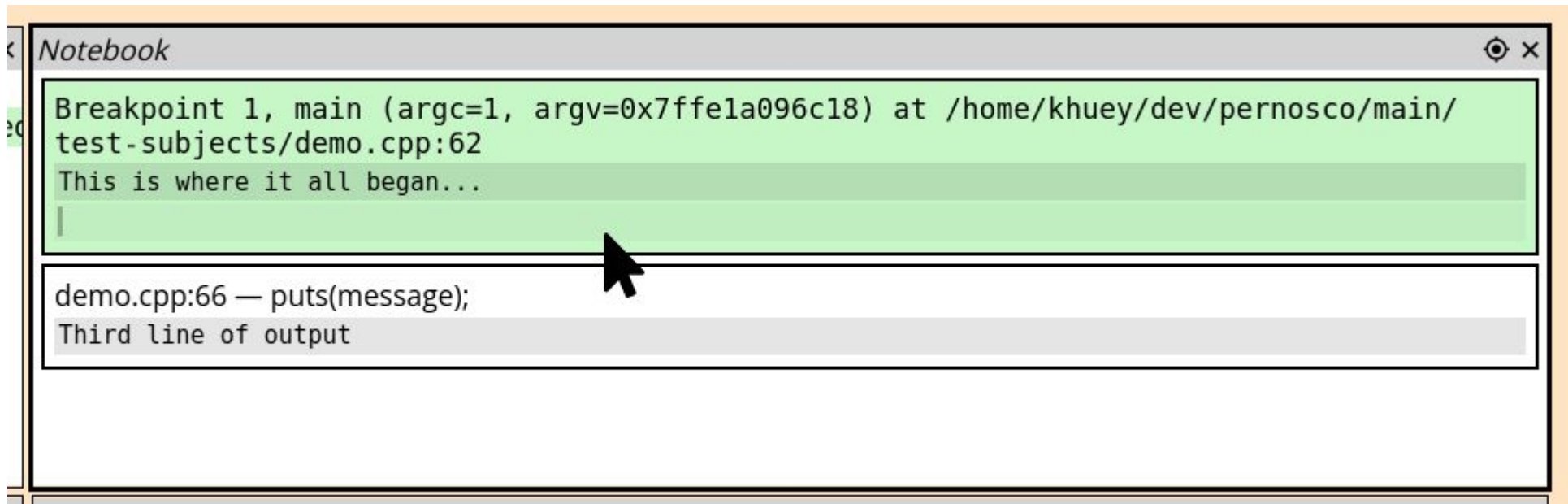
▼ Process 63340 /home/khuey/dev/pernosco/main/test-subjects/obj/bin/demo
Thread 63340 (demo)

Stack of selected thread (thread 63340 (demo))

__libc_start_main (...) at libc-start.c:308
main (...) at demo.cpp:63

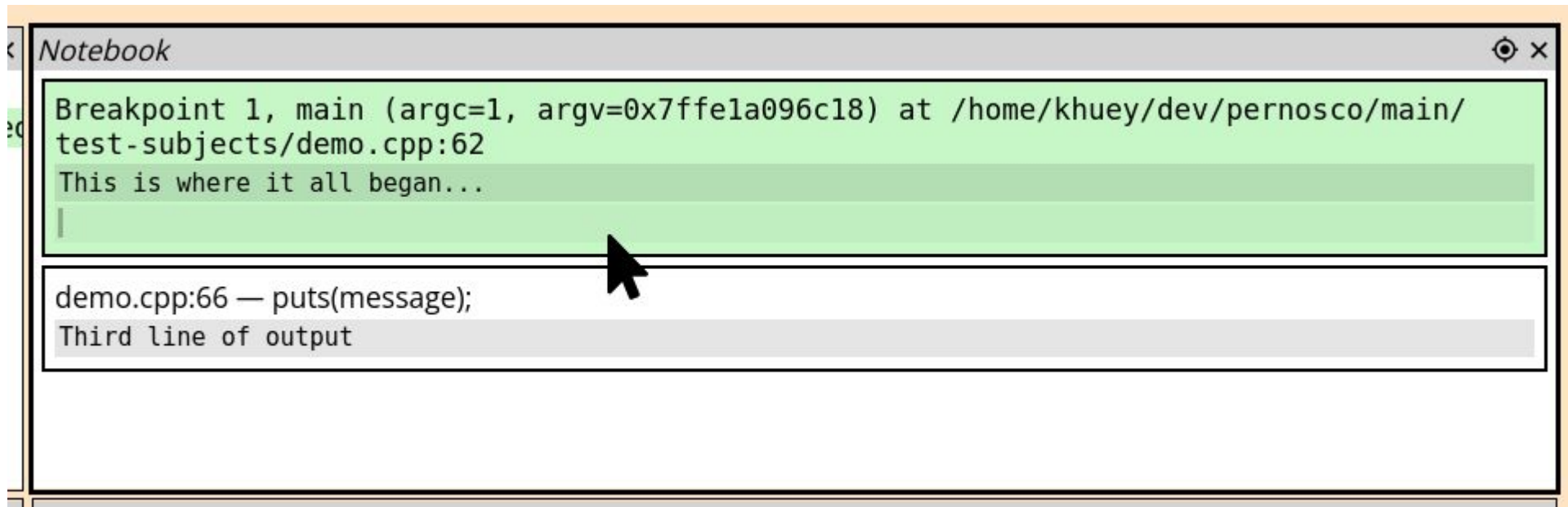
Support user habits:

Debugging notebook



Thin Web client, server state

→ Easy collaboration



The screenshot shows a web-based notebook interface with a title bar labeled "Notebook". The main content area is divided into two sections. The top section has a light green background and contains the text: "Breakpoint 1, main (argc=1, argv=0x7ffe1a096c18) at /home/khuey/dev/pernosco/main/test-subjects/demo.cpp:62" followed by "This is where it all began..." and a cursor on a new line. The bottom section has a light gray background and contains the text: "demo.cpp:66 — puts(message);" followed by "Third line of output". A mouse cursor is pointing at the boundary between the two sections.


```
Notebook
```

Breakpoint 1, main (argc=1, argv=0x7ffe1a096c18) at /home/khuey/dev/pernosco/main/test-subjects/demo.cpp:62
This is where it all began...
|

demo.cpp:66 — puts(message);
Third line of output

Unify paradigms

“Breakpoints and logging are kind of the same”



```
/builds/worker/workspace/build/src/layout/generic/nsBlockFrame.cpp
1138 // our floats list, since a first-in-flow might get pushed to
1139 // continuation of its containing block. But it's not permitted
1140 // outside that time.
1141 nsLayoutUtils::AssertNoDuplicateContinuations(this, mFloats);
1142 #endif
1143
1144 // ALWAYS drain overflow. We never want to leave the previnflow
1145 // overflow lines hanging around; block reflow depends on the
1146 // overflow line lists being cleared out between reflow passes.
1147 DrainOverflowLines();
1148
1149 bool blockStartMarginRoot, blockEndMarginRoot;
1150 IsMarginRoot(&blockStartMarginRoot, &blockEndMarginRoot);
1151
1152 // Cache the consumed height in the block reflow input so that
1153 // to continually recompute it.
1154 BlockReflowInput state(*reflowInput, aPresContext, this, block
1155                        blockEndMarginRoot, needFloatManager, c
1156
1157 if (GetStateBits() & NS_BLOCK_NEEDS_BIDI_RESOLUTION)
1158     static_cast<nsBlockFrame*>(FirstContinuation())->ResolveBidi
1159
1160 // Handle paginated overflow (see nsContainerFrame.h)
1161 nsOverflowAreas ocBounds;
1162 nsReflowStatus ocStatus;
1163 if (GetPrevInFlow()) {
1164     ReflowOverflowContainerChildren(aPresContext, *reflowInput,
1165                                     ocStatus);
1166 }
1167
1168 // Now that we're done cleaning up our overflow container list
1169 // give |state| its nsOverflowContinuationTracker.
1170 nsOverflowContinuationTracker tracker(this, false);
```

Unify paradigms

The screenshot displays a GDB debugging session for Firefox. The interface is divided into three main sections:



- Top Section:** Contains the source code of `nsGlobalWindowInner.cpp`. The code includes a warning log and a function `nsGlobalWindowInner::Alert` that writes to `stdout/stderr`. The warning log indicates a failure with result `0x80004005` in `nsPermissionManager.cpp`.
- Bottom Left Section:** Shows the stack of the selected thread (thread 1975, Firefox-bin). The stack trace includes frames for `__libc_start_main`, `main`, `do_main`, `nsBootstrapImpl::XRE_main`, `XRE_main`, `XREMain::XRE_main`, `ScopedXPCOMStartup::Initialize`, and `NS_InitXPCOM`.
- Bottom Right Section:** Displays the GDB console output for process 1975. The output shows the execution of `NS_InitXPCOM` and a breakpoint hit at `XPComInit.cpp:254`. The console also shows the elapsed time and the state of the thread.

The GDB console output includes the following text:



```
Gdb process 1975 /builds/worker/workspace/build/application/firefox/firefox -marionette -profile /tmp/tmpLLp_ag.mozrunner
(pernosco) elapsed-time
Elapsed Time (s): 31.184074678
(pernosco) break NS_InitXPCOM
Breakpoint 1 at 0x7f0f25724a63: file /builds/worker/workspace/build/src/xpcom/build/XPComInit.cpp, line 254.
(pernosco) reverse-continue
Continuing.
[New Thread 1975.2115]

Thread 1 hit Breakpoint 1, NS_InitXPCOM (aResult=0x7f0f36f87770, aBinDirectory=0x7f0f36f59c40, aAppFileLocationProvider=0x7ffcef6f76f8) at /builds/worker/workspace/build/src/xpcom/build/XPComInit.cpp:254
(pernosco) elapsed-time
Elapsed Time (s): 0.19934826034
(pernosco) # We just jumped backwards through 30 seconds of real time.
(pernosco)
```

Composability

Writes to stdout/stderr  

Line A
Line B
Line C

Executions of *_IO_puts* *No condition* *No print*  

_IO_puts (str@0x5598ae6f2004="Line A") = 7
_IO_puts (str@0x5598ae6f200b="Line B") = 7
_IO_puts (str@0x5598ae6f2012="Line C") = 7

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!