FUNCTION CALL TRACING

XRAY IN LLVM

Dean Berris (@deanberris)

WHAT TO EXPECT

WHAT, HOW, AND WHEN.

TOPICS TO DISCUSS

- What is XRay?
- How do you use XRay?
- How is XRay implemented?
- What to expect next?

BEFORE ANYTHING ELSE...

WHY XRAY?

LATENCY & PERFORMANCE DEBUGGING IS HARD.

- Sampling is an OK approximation for *mean/average* latency, but not for tail latency debugging.
- Existing solutions require some OS-level support, special privileges, and non-trivial overheads.
- Manual tracing/logging is not cheap and usually error-prone.

XRAY: HIGH LEVEL FEATURES AND CONSTRAINTS

- XRay leverages compiler knowledge of code structure to provide instrumentation in interesting places.
- XRay instrumentation points are cheap, can be deployed in production with little interference/effect.
- Dynamic control (being able to turn things on/off at runtime) is crucial for production deployments.
- Gathering data that can be analysed offline.

A WALKTHROUGH

WHAT IS XRAY?

Image from https://upload.wikimedia.org/wikipedia/commons/c/cd/Historical_X-ray_nci-vol-1893-300.jpg

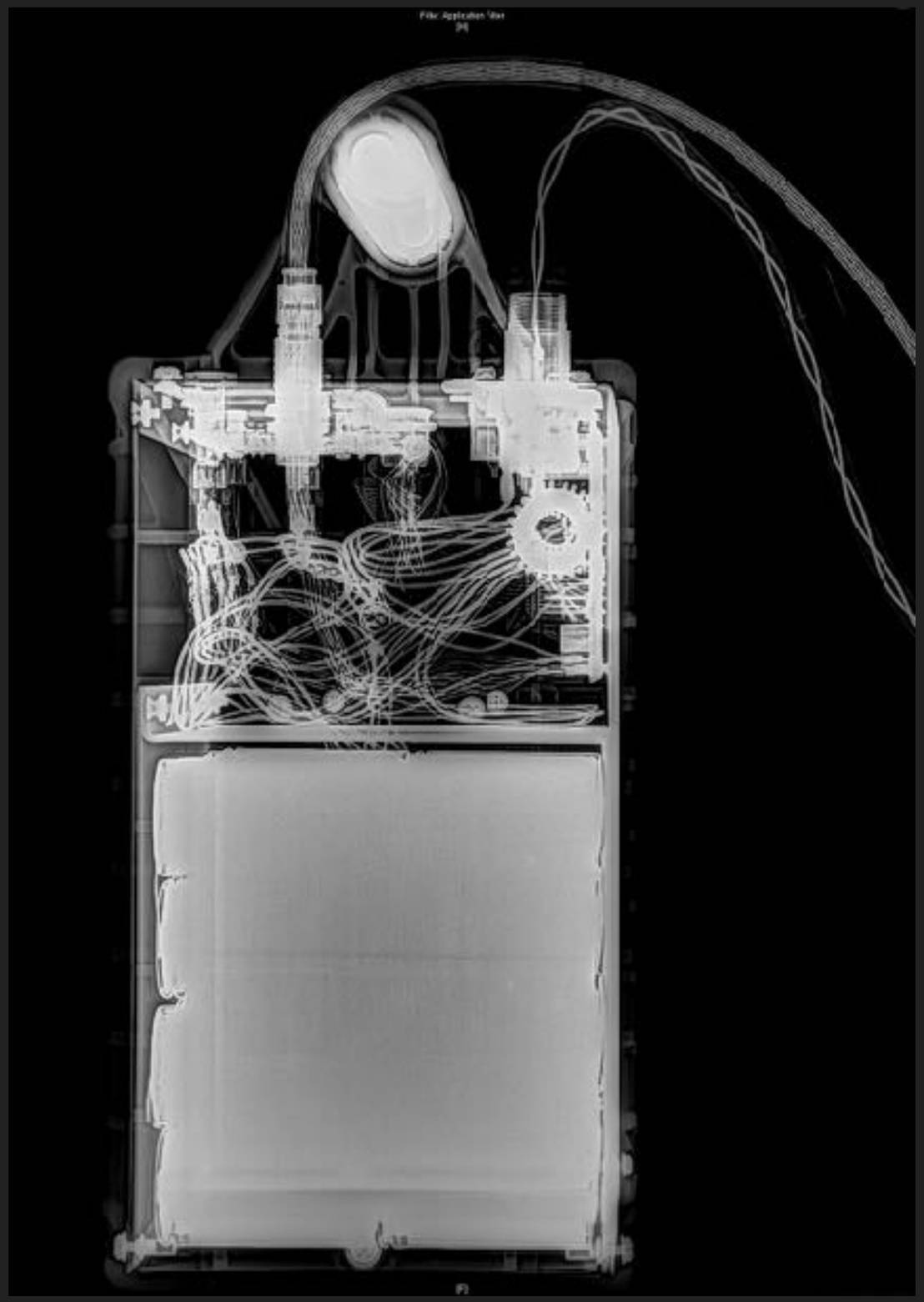
WHAT IS XRAY?

- Compiler-inserted instrumentation in functions
- Runtime library for dynamic instrumentation
- Tools for analysing traces

COMPILER INSERTED INSTRUMENTATION.

NO-OPS IN THE RIGHT PLACES

- We add "sleds" of instructions that we can overwrite at runtime.
- We put them in strategic places, to allow us to catch entry and exits at runtime.
- We put them on "interesting" functions, according to number of instructions.
- We also put them on explicitly marked functions.



AN EXAMPLE IN C++

CODE.

```
[[clang::xray_always_instrument]] void foo() { printf("Hello, XRay!\n"); }
```

Explicitly annotated functions for "always" or "never" instrumented functions.

clang++ **-fxray-instrument** ...

```
  .globl  _Z3foov
  .p2align  4, 0x90
  .type _Z3foov,@function
_Z3foov:                                # @_Z3foov
.Lfunc_begin0:
  .file 22 "test.cc"
  .loc  22 6 0                          # test.cc:6:0
  .cfi_startproc
# BB#0:                                 # %entry
  .p2align  1, 0x90
.Lxray_sled_0:
  .ascii  "\353\t"                      Entry sled.
  nopw  512(%rax,%rax)
.Ltmp0:
  pushq %rbp
.Ltmp1:
  .cfi_def_cfa_offset 16
.Ltmp2:
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
.Ltmp3:
  .cfi_def_cfa_register %rbp
  subq  $16, %rsp
  movabsq $.L.str, %rdi
.Ltmp4:
  .loc  22 6 48 prologue_end    # test.cc:6:48
  movb  $0, %al
  callq printf
  .loc  22 6 74 is_stmt 0       # test.cc:6:74
  movl  %eax, -4(%rbp)          # 4-byte Spill
  addq  $16, %rsp
  popq  %rbp
  .p2align  1, 0x90
.Lxray_sled_1:
  retq                                  Exit sled.
  nopw  %cs:512(%rax,%rax)
.Ltmp5:
.Lfunc_end0:
  .size _Z3foov, .Lfunc_end0-_Z3foov
  .cfi_endproc
```

WORKS WITH LLVM IR.

```
; Function Attrs: uwtable
define void @_Z3foov() #0 {
entry:
  %call = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([14 x i8], [14 x i8]* @.str, i32 0, i32 0))
  ret void
}
```

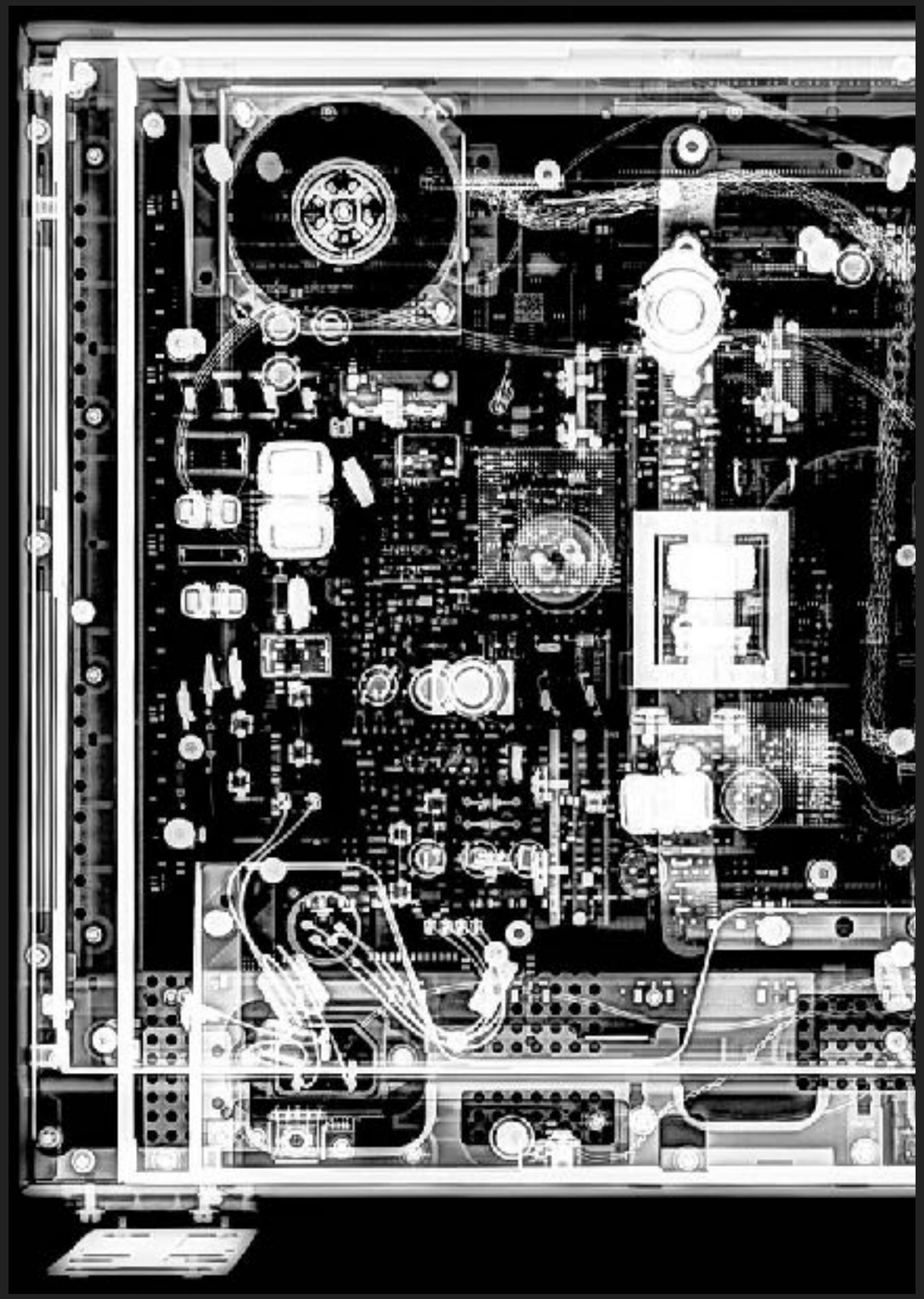
```
attributes #0 = { uwtable "disable-tail-calls"="false" "function-instrument"="xray-always" "less-precise-fpmad"="false" "no-frame-pointer-elim"="
true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"
="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

"function-instrument" is the keyword attribute coming from source-level attributes in C/C++, may be either **"xray-always"** or **"xray-never"**.

automatic function size heuristic comes in with **"xray-instruction-threshold"**="NNN".

THE INSTRUMENTATION MAP

- At runtime we need to know where the sleds are, and what kind of sleds they are in the binary.
- We keep track of those in the object files.
- We also know whether to always or sometimes instrument them.
- We keep some bytes for future use later, and to align the entries better.
- They are concatenated together by the linker.



THE INSTRUMENTATION MAP

```
  .p2align  4, 0x90
  .quad .Lxray_synthetic_0
  .section  xray_instr_map,"a",@progbits
.Lxray_synthetic_0:
  .quad .Lxray_sled_0
  .quad _Z3foov
  .byte 0
  .byte 1
  .zero 14
  .quad .Lxray_sled_1
  .quad _Z3foov
  .byte 1
  .byte 1
  .zero 14
  .text
```

Our XRay instrumentation map section.

Our entry sled for function "void foo()", which should always be instrumented.

Our exit sled for function "void foo()", which should always be instrumented.

RUNTIME LIBRARY.

PATCHING/UNPATCHING

- Read through the instrumentation map.
- Compute the function id (function appearance order) and get the offset of the __xray_Function{Enter,Exit} from the sled.
- For entry sleds, make them look like:
  "mov <function id>, %r10d; call __xray_FunctionEnter"
- For exit sleds, make them look like:
  "mov <function id>, %r10d; jmp __xray_FunctionExit"

BEFORE PATCHING

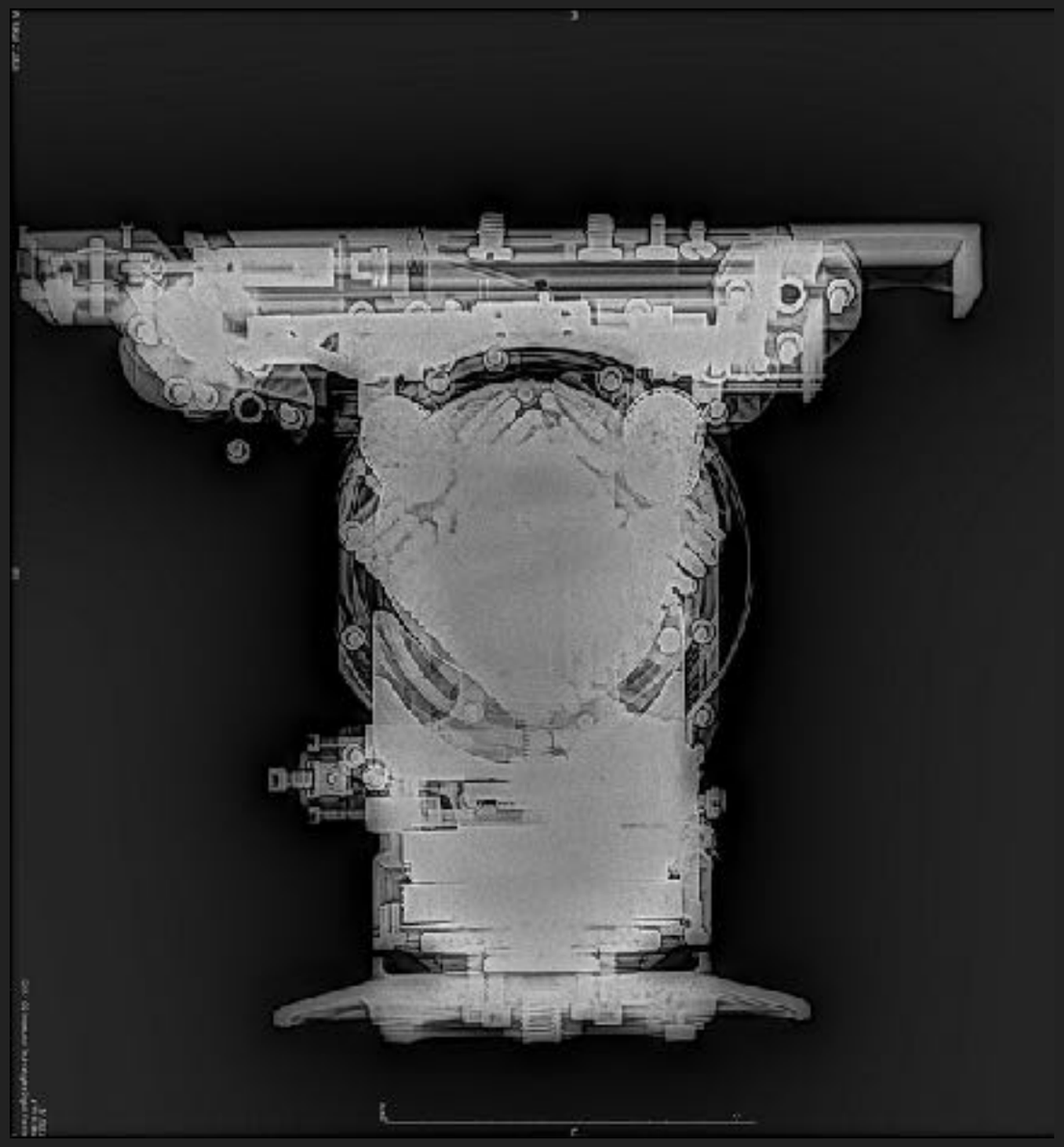
```
(gdb) disassemble foo
Dump of assembler code for function foo():
   0x0000000000415320 <+0>:     jmp    0x41532b <foo()+11>
   0x0000000000415322 <+2>:     nopw   0x200(%rax,%rax,1)
   0x000000000041532b <+11>:    push   %rbp
   0x000000000041532c <+12>:    mov    %rsp,%rbp
   0x000000000041532f <+15>:    sub    $0x10,%rsp
   0x0000000000415333 <+19>:    movabs $0x41ab2c,%rdi
   0x000000000041533d <+29>:    mov    $0x0,%al
   0x000000000041533f <+31>:    callq  0x401a50 <printf@plt>
   0x0000000000415344 <+36>:    mov    %eax,-0x4(%rbp)
   0x0000000000415347 <+39>:    add    $0x10,%rsp
   0x000000000041534b <+43>:    pop    %rbp
   0x000000000041534c <+44>:    retq
   0x000000000041534d <+45>:    nopw   %cs:0x200(%rax,%rax,1)
End of assembler dump.
```

AFTER PATCHING

```
(gdb) disassemble foo
Dump of assembler code for function foo():
   0x0000000000415320 <+0>:     mov    $0x1,%r10d
   0x0000000000415326 <+6>:     callq  0x4145e0 <__xray_FunctionEntry>
   0x000000000041532b <+11>:    push   %rbp
   0x000000000041532c <+12>:    mov    %rsp,%rbp
   0x000000000041532f <+15>:    sub    $0x10,%rsp
   0x0000000000415333 <+19>:    movabs $0x41ab2c,%rdi
   0x000000000041533d <+29>:    mov    $0x0,%al
   0x000000000041533f <+31>:    callq  0x401a50 <printf@plt>
   0x0000000000415344 <+36>:    mov    %eax,-0x4(%rbp)
   0x0000000000415347 <+39>:    add    $0x10,%rsp
   0x000000000041534b <+43>:    pop    %rbp
   0x000000000041534c <+44>:    mov    $0x1,%r10d
   0x0000000000415352 <+50>:    jmpq   0x4146d0 <__xray_FunctionExit>
End of assembler dump.
```

LOGGING FUNCTION ENTRY/EXITS

- The provided logging function gets called from the trampolines.
- The default implementation logs the timestamp (TSC), thread id, the cpu id, and whether it was a function entry or an exit.
- We can then reconstruct the function call stack offline based on the entry/exit events in the log.



CUSTOM FUNCTIONALITY

- We can install any handler with `__xray_set_handler(...)`.
  - The handler only needs to take two arguments: function id (int32_t) and entry type (an enum type, i.e. int).
  - The handler has to be thread-safe.
  - The handler can pretty much do what it wants. :)
- Install other handlers with `__xray_set_handler_arg1(...)` for capturing first argument of functions attributed to capture the first argument (useful for "this" pointer)
- Support for custom event logging with `__xray_set_customevent_handler(...)` for capturing custom events provided by `__xray_customevent(...)` clang built-in.

ANALYSIS TOOLS:

STATISTICS AND RECONSTRUCTION

- Analysis tools are currently part of the LLVM tools distribution. The tool is called 'llvm-xray' which includes:
  - Function call accounting statistics.
  - Function call graph with latency distributions.
  - Function call stacks with latency sums and counts.

PRACTICAL MATTERS

HOW TO USE XRAY TODAY

USING XRAY IN LLVM

- Top of Trunk Clang + LLVM + compiler-rt
- Top of Trunk LLVM, by adding attributes to LLVM IR
- Available for Linux running on x86_64, ARM7 and ARM8 (no thumb), AArch64, PowerPC 64 (little endian), MIPS

DOCUMENTATION FOR XRAY

- XRay white paper
- High level documentation (llvm.org/docs/XRay.html)
- Debugging Example (llvm.org/docs/XRayExample.html)

AN EXAMPLE IN C++

CODE AND TOOLS

HELLO, XRAY! (CODE)

```
#include <iostream>

[[clang::xray_always_instrument]] void foo() {
  std::cout << "Hello, XRay!" << std::endl;
}

[[clang::xray_always_instrument, clang::xray_log_args(1)]]
void bar(int i) {
  std::cout << "Captured: " << i << std::endl;
}

[[clang::xray_always_instrument]] int main(int argc, char* argv[]) {
  foo();
  bar(argc);
}
```

HELLO, XRAY! (BUILD)

```
clang++ -fxray-instrument -std=c++11 \
    hello_xray.cpp -o hello_xray
```

HELLO, XRAY! (RUN)

```
$ ./hello_xray
Hello, XRay!
Captured: 1
```

```
$ XRAY_OPTIONS="patch_premain=true" ./hello_xray
==NNNNN==XRay: Log file in 'xray-log.hello_xray.xxxx'
Hello, XRay!
Captured: 1
```

HELLO, XRAY! (TOOLS)

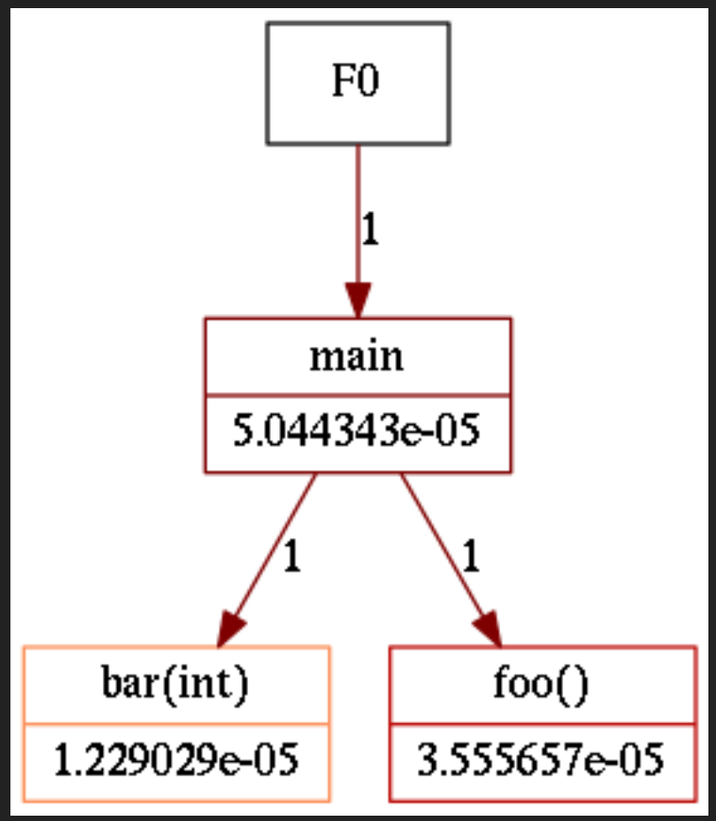
```
$ llvm-xray stack -instr_map=./hello_xray xray-log.hello_xray.*
Unique Stacks: 2
Top 10 Stacks by leaf sum:

Sum: 124448
lvl   function                                                  count            sum
#0    main                                                          1         176552
#1    foo()                                                         1         124448

Sum: 43016
lvl   function                                                  count            sum
#0    main                                                          1         176552
#1    bar(int)                                                      1          43016


Top 10 Stacks by leaf count:

Count: 1
lvl   function                                                  count            sum
#0    main                                                          1         176552
#1    foo()                                                         1         124448

Count: 1
lvl   function                                                  count            sum
#0    main                                                          1         176552
#1    bar(int)                                                      1          43016
```

HELLO, XRAY! (TOOLS)

```
$ llvm-xray account -instr_map=./hello_xray xray-log.hello_xray.*
Functions with latencies: 3
   funcid      count [      min,      med,      90p,      99p,      max]      sum  function
   1           1 [ 0.000036,  0.000036,  0.000036,  0.000036,  0.000036]  0.000036  <invalid>:0:0: foo()
   2           1 [ 0.000012,  0.000012,  0.000012,  0.000012,  0.000012]  0.000012  <invalid>:0:0: bar(int)
   3           1 [ 0.000050,  0.000050,  0.000050,  0.000050,  0.000050]  0.000050  <invalid>:0:0: main
```

HELLO, XRAY! (TOOLS)

```
$ llvm-xray graph -instr_map=./hello_xray xray-log.hello_xray.* -color-edges=count \
    -edge-label=count -color-vertices=sum -vertex-label=sum | dot -png -x > /tmp/hello_xray.png
```



HELLO, XRAY! (TOOLS)

```
$ llvm-xray stack -instr_map=./hello_world_fdr \
    xray-log.hello_world_fdr.s3c4ft --keep-going --stack-format=flame
-all-stacks 2>/dev/null | ~/Source/FlameGraph/flamegraph.pl \
    > /tmp/hello_xray_flame.svg
```

FlameGraph tool from
https://github.com/brendangregg/FlameGraph

DEMO.

FOR THE DEVS.

XRAY IMPLEMENTATION DETAILS

XRAY IN LLVM

LLVM

- XRay Instrumentation Pass

Takes MachineFunctions and inserts pseudo instructions marking where the start of the function is, where the exits are, and any custom event points.

- CodeGen implementations for targets

Lowering to per-target specific assembly, ELF/MachO sections for the instrumentation map.

- Analysis tools and libraries

Implementation of the llvm-xray tool with various sub-commands (extract, convert, stacks, account, graph, graph-diff).

Trace reader library for reading XRay traces.

InstrumentationMap class for working with the instrumentation map in XRay-instrumented binaries.

XRAY IN CLANG

- Driver Options

Support for -fxray-instrument, -fxray-instruction-threshold, -fxray-always-instrument=..., and -fxray-never-instrument=... -- also adding the compiler runtime library when linking binaries.

- Builtin Support

Support for __xray_customevent(...).

- Source-level Attributes (C/C++)

We use C++ attributes, [[clang::xray_always_instrument]], [[clang::xray_never_instrument]], and [[clang::xray_log_args(N)]] and __attribute__((...)) analogues of these.

XRAY IN COMPILER-RT

- XRay APIs and Implementation

Control APIs for installing handlers, turning instrumentation on/off, etc.

- Logging Implementations

Basic (naive) mode and Flight Data Recorder mode.

SUPPORT MATRIX

feature	x86_64	ppc64le	arm	aarch64	mips
no-arg logging	✔	✔	✔	✔	✔
1-arg logging	✔	✔	✔	✔	✔
basic/naive mode	✔	✔	✔	✔	✔
custom events	✔				
flight data recorder mode	✔				

WHAT TO EXPECT NEXT.

MORE TO COME

- More Analysis Tools
- Support for more function exit types (exceptional exits, interrupts (maybe), etc.)
- More architectures and platforms/OSes

THANK YOU!

QUESTIONS?

Dean Berris (@deanberris), XRay Team (google-xray@googlegroups.com)