Arbitrary Instruction Tracing with DTrace

Christos Margiolis
christos@FreeBSD.org

March 24, 2024
AsiaBSDCon 2024 — Taipei, Taiwan
DTrace quick background

- Dynamic tracing framework.
- Observe kernel behavior in real-time.
- Provider: Module that performs a particular instrumentation in the kernel.
- Probe: Specific point of instrumentation.
- D language.
- https://illumos.org/books/dtrace
The FBT provider

▶ Trace the entry and return points of a kernel function.
▶ Cannot trace specific instructions and inline functions.

```bash
# dtrace -n 'fbt::malloc:entry {printf("%s", execname);}'
dtrace: description 'fbt::malloc:entry ' matched 1 probe
CPU ID FUNCTION:NAME
  3 30654 malloc:entry dtrace
  0 30654 malloc:entry pkg
  1 30654 malloc:entry Xorg
  3 30654 malloc:entry firefox
  2 30654 malloc:entry zfskern
  3 30654 malloc:entry kernel
^C
```
The kinst provider

- Inspired by FBT.
- Trace arbitrary machine instructions in a kernel function.
- Trace inline functions.
- More fine-grained tracing (specific if statements, loops, branches, ...). Requires C-to-Assembly translation skills.
- Available on amd64, arm64 and riscv.
- In the future: build higher-level tooling, detect and put return probes on tail-call optimized functions.
- sys/cddl/dev/kinst/
# dtrace -n 'kinst::amd64_syscall:'
dtrace: description 'kinst::amd64_syscall:' matched 458 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>80676</td>
<td>amd64_syscall:323</td>
</tr>
<tr>
<td>2</td>
<td>80677</td>
<td>amd64_syscall:326</td>
</tr>
<tr>
<td>2</td>
<td>80678</td>
<td>amd64_syscall:334</td>
</tr>
<tr>
<td>2</td>
<td>80679</td>
<td>amd64_syscall:339</td>
</tr>
<tr>
<td>2</td>
<td>80680</td>
<td>amd64_syscall:345</td>
</tr>
<tr>
<td>2</td>
<td>80681</td>
<td>amd64_syscall:353</td>
</tr>
</tbody>
</table>

^C
# kgdb
(kgdb) disas /r vm_fault
Dump of assembler code for function vm_fault:
  0xffffffff80876df0 <+0>:  55  push  %rbp
  0xffffffff80876df1 <+1>:  48 89 e5  mov  %rsp,%rbp
  0xffffffff80876df4 <+4>:  41 57  push  %r15

# dtrace -n 'kinst::vm_fault:4 {printf("%#x", regs[R_RSI])}
};'
  2  81500   vm_fault:4  0x827c56000
  2  81500   vm_fault:4  0x827878000
  2  81500   vm_fault:4  0x1fab9bef0000
  2  81500   vm_fault:4  0xe16cf749000
  0  81500   vm_fault:4  0x13587c366000
^C
# dtrace -n 'kinst::critical_enter:return'
dtrace: description 'kinst::critical_enter:return' matched
  130 probes

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>71024</td>
<td>spinlock_enter:53</td>
</tr>
<tr>
<td>0</td>
<td>71024</td>
<td>spinlock_enter:53</td>
</tr>
<tr>
<td>1</td>
<td>70992</td>
<td>uma_zalloc_arg:49</td>
</tr>
<tr>
<td>1</td>
<td>70925</td>
<td>malloc_type_zone_allocated:21</td>
</tr>
<tr>
<td>1</td>
<td>70994</td>
<td>uma_zfree_arg:365</td>
</tr>
<tr>
<td>1</td>
<td>70924</td>
<td>malloc_type_freed:21</td>
</tr>
<tr>
<td>0</td>
<td>71024</td>
<td>spinlock_enter:53</td>
</tr>
<tr>
<td>0</td>
<td>70947</td>
<td>_epoch_enter_preempt:122</td>
</tr>
<tr>
<td>0</td>
<td>70949</td>
<td>_epoch_exit_preempt:28</td>
</tr>
<tr>
<td>0</td>
<td>71024</td>
<td>spinlock_enter:53</td>
</tr>
<tr>
<td>0</td>
<td>71024</td>
<td>spinlock_enter:53</td>
</tr>
<tr>
<td>0</td>
<td>70947</td>
<td>_epoch_enter_preempt:122</td>
</tr>
<tr>
<td>0</td>
<td>70949</td>
<td>_epoch_exit_preempt:28</td>
</tr>
</tbody>
</table>

^C
High-level ideas

- How are instructions instrumented?
- Architecture-dependent code.
- Inline function tracing.
Instruction instrumentation

- Probe information is passed from dtrace(1) to libdtrace to kinst(4) using a character device file in /dev/dtrace/kinst.
- kinst disassembles the function and creates probes for each of the target instructions.
- The original instruction is overwritten with a breakpoint instruction.
- When the CPU hits the breakpoint, we jump into kinst_invop() through the trap handler.
- kinst decides if the instruction is to be emulated or executed in a trampoline.
- Trace the instruction and continue execution.
Instruction instrumentation

- dtrace(1) command
- libdtrace: ioctl with probe info to /dev/dtrace/kinst
- kinst(4): create probe(s), overwrite instruction(s) with breakpoints
Instruction instrumentation

```
func

instr

instr -> breakpoint
... 
... 

breakpoint handler -> dtrace_invop() -> kinst_invop()

instr
```
Trampoline: Overview

- Emulating every single instruction for each architecture is tedious and error prone.
- Target instruction is copied there and execution is transferred to the trampoline manually.
- How do we return back?
Trampoline: Under the hood

- Executable block of memory.
- Memory "chunks" of size \texttt{PAGE\_SIZE} stored in a TAILQ
- \texttt{vm\_map\_find(9)} with \texttt{VM\_PROT\_EXECUTE}, \texttt{vm\_map\_remove(9)}, \texttt{kmem\_back(9)}, \texttt{kmem\_unback(9)}, \texttt{malloc(9)}.
- Allocated above \texttt{KERNBASE} (amd64), or \texttt{VM\_MIN\_KERNEL\_ADDRESS} (rest).
- Logically divided into individual trampolines using \texttt{BITSET(9)}.
- \texttt{kinst\_trampoline\_alloc()} finds and returns the next free trampoline.
Trampoline: Under the hood

- **amd64**: Per-thread and per-CPU trampolines.
- **arm64** and **riscv**: Per-probe trampolines.
Trampoline: amd64 control flow (to be deprecated)

The per-thread/per-CPU trampolines are rewritten upon every instrumentation. Will be replaced by the arm64/riscv mechanism and use per-probe trampolines to avoid race bugs.
Trampoline: arm64/riscv control flow

Synchronization is done through a DPCPU(9) kinst_cpu_state structure.
Architecture-specific caveats

- **amd64**
  - The ISA is... complicated. Lots of "special" instructions to handle during disassembly parsing.
  - RIP-relative instructions have to have their displacements re-encoded to be relative to the trampoline in order to be executed in a trampoline.
  - `call` instructions have to be emulated in assembly (see `bp_call label` in `sys/cddl/dev/dtrace/amd64/dtrace_asm.S`).

- **arm64, riscv**
  - Unlike amd64, encoding trampoline-relative offsets for PC-relative instructions is not possible in a single instruction. All PC-relative instructions have to be emulated in `kinst_emulate()`.

- Some functions and instructions are unsafe to trace (listed in man page).
Inline function tracing

- Syntax: `kinst::<inline_func>::<entry|return>`
- All the hard work is done in libdtrace, instead of `kinst(4)`.  
- Uses the DWARF and ELF standards.
  - If the function is an inline, libdtrace calculates the function boundaries and offsets and creates regular kinst probes for each one of the inline copies found.
  - If the function is *not* an inline, the probe is converted to an FBT one, to avoid code duplication.

- Done for each loaded kernel module. Painfully slow...
- Can handle nested inline functions.
- `cddl/contrib/opensolaris/lib/libdtrace/common/dt_sugar.c`
- https://margiolis.net/w/dwarf_inline/
- https://margiolis.net/w/kinst_inline/
Inline function tracing

Inline function

```c
kinst::cam_iosched_has_more_trim:entry
{
    printf("\t%d\t%s", pid, execname);
}
```

```c
kinst::cam_iosched_get_trim:13,
kinst::cam_iosched_next_bio:13,
kinst::cam_iosched_schedule:40
{
    printf("\t%d\t%s", pid, execname);
}
```

Non-inline function

```c
kinst::malloc:entry
{
    exit(0);
}
```

```c
fbt::malloc:entry
{
    exit(0);
}
```
Inline function tracing

```
# dtrace -n 'kinst::vm_page_mvqueue:entry,kinst::
    vm_page_mvqueue:return'
```
Inline function tracing: DWARF

- Debugging information is represented as a tree of entries, one tree per compilation unit. Entries correspond to functions, variables, arguments, etc.
- Entries are called DIE (I know...); Debugging Information Entry.
- Each DIE has various attributes (name, location, file, ...).
- Functions that get inlined have an ”inlined” attribute set.
Inline function tracing: DWARF

Inline function declaration entry:

<1><1dfa144>: Abbrev Number: 94 (DW_TAG_subprogram)
   <1dfa145> DW_AT_name : (indirect string)
         vfs_freevnodes_dec
   <1dfa149> DW_AT_decl_file : 1
   <1dfa14a> DW_AT_decl_line : 1447
   <1dfa14c> DW_AT_prototyped : 1
   <1dfa14c> DW_AT_inline : 1

What we care about:

- 0x1dfa144: DIE address
- DW_TAG_subprogram: Function
- DW_AT_inline: Inlined
- DW_AT_name: Function name
Inline function tracing: DWARF

Inline copy entry:

<3><1dfe45e>: Abbrev Number: 24 (DW_TAG_inlined_subroutine)
<1dfe45f> DW_AT_abstract_origin: <0x1dfa144>
<1dfe463> DW_AT_low_pc : 0xffffffff80cf701d
<1dfe46b> DW_AT_high_pc : 0x38
<1dfe46f> DW_AT_call_file : 1
<1dfe470> DW_AT_call_line : 3458
<1dfe472> DW_AT_call_column : 5

What we care about:

- DW_TAG_inlined_subroutine: Inline copy
- DW_AT_abstract_origin: Points to 0x1dfa144.
- DW_AT_lowpc: Location in memory.
- DW_AT_highpc: Size.
Inline function tracing: Calculating call boundaries

Two ways to define locations:
- DW_AT_lowpc and DW_AT_highpc.
- DW_AT_ranges.
Inline function tracing: Calculating call boundaries

Listing 1: DIE with low and high PC boundaries

- lower_bound = low_pc
- upper_bound = low_pc + high_pc
Inline function tracing: Calculating call boundaries

- **DW_AT_ranges** refers to the `.debug_ranges` section of debug files.
- Usually means that the inline copy has been split into multiple different locations in memory (early returns).
- We end up with more than one return probe.

Listing 2: DIE with DW_AT_ranges boundaries
Inline function tracing: Calculating call boundaries

Look for range at 0x1f1290:

```
# dwarfdump -N dbgfile
...
Ranges group 38809:
ranges: 3 at .debug_ranges offset 2036368 (0x001f1290) (48 bytes)
[ 0] range entry 0x000025c8 0x000025f9
[ 1] range entry 0x0000261a 0x00002621
...
```

- Values relative to the root (compilation unit) DIE’s DW_AT_lowpc.
Inline function tracing: Calculating call boundaries

<0><1dee9fb>: Abbrev Number: 1 (DW_TAG_compile_unit)
  ...
  <1deea02>   DW_AT_name : (indirect string) /usr/src/sys/kern/vfs_subr.c
  ...
  <1deea0e>   DW_AT_low_pc : 0xffffffff80cf4020
  <1deea16>   DW_AT_high_pc : 0xde3d

Listing 3: Compilation unit DIE
Inline function tracing: Calculating call boundaries

First range location:

\[
Loc_{1_{\text{lower}}} = Root_{\text{lower}} + Rng_{1_{\text{lower}}}
\]

\[
Loc_{1_{\text{upper}}} = Root_{\text{lower}} + Rng_{1_{\text{upper}}}
\]

Second range location:

\[
Loc_{2_{\text{lower}}} = Root_{\text{lower}} + Rng_{2_{\text{lower}}}
\]

\[
Loc_{2_{\text{upper}}} = Root_{\text{lower}} + Rng_{2_{\text{upper}}}
\]

- \( Loc_{1_{\text{lower}}} \) is the entry point of the inline function.
- \( Loc_{1_{\text{upper}}} \) and \( Loc_{2_{\text{upper}}} \) are the return points.
Inline function tracing: Finding the caller function

For each inline copy:

- We know its boundaries.
- Need to scan ELF information to figure out which function the copy is inlined in.

The name of the caller function corresponds to the name of the symbol satisfying the following condition:

\[ Sym_{\text{lower}} \leq Inl_{\text{lower}} \leq Inl_{\text{upper}} \leq Sym_{\text{upper}} \]
Inline function tracing: entry and return offsets

The entry and return offsets are then calculated as:

\[
Entry = \text{lnl}_{\text{lower}} - \text{caller}_{\text{lower}}
\]

\[
Return = \text{lnl}_{\text{upper}} - \text{caller}_{\text{lower}}
\]

- Not exactly that simple, but out of scope. Please see https://margiolis.net/w/kinst_inline/#heuristic-entry-return
- 1e136a9cbd3a added a -d option to dtrace(1) to able to dump the resulting D script.
Inline function tracing

And **FINALLY**...
Inline function tracing

```
# dtrace -n 'kinst::vm_page_mvqueue:entry,kinst::vm_page_mvqueue:return'

<table>
<thead>
<tr>
<th>CPU</th>
<th>ID</th>
<th>FUNCTION:NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>95381</td>
<td>vm_page_activate:13</td>
</tr>
<tr>
<td>3</td>
<td>95389</td>
<td>vm_page_activate:146</td>
</tr>
<tr>
<td>2</td>
<td>95381</td>
<td>vm_page_activate:13</td>
</tr>
<tr>
<td>2</td>
<td>95389</td>
<td>vm_page_activate:146</td>
</tr>
<tr>
<td>1</td>
<td>95387</td>
<td>vm_page_advise:332</td>
</tr>
<tr>
<td>1</td>
<td>95400</td>
<td>vm_page_advise:421</td>
</tr>
<tr>
<td>1</td>
<td>95387</td>
<td>vm_page_advise:332</td>
</tr>
<tr>
<td>1</td>
<td>95400</td>
<td>vm_page_advise:421</td>
</tr>
<tr>
<td>1</td>
<td>95387</td>
<td>vm_page_advise:332</td>
</tr>
</tbody>
</table>
```

^C
Inline function tracing

- D38825
- D39259
- D40874
Acknowledgments

Mark Johnston <markj@FreeBSD.org>
Mitchell Horne <mhorne@FreeBSD.org>
References


- FreeBSD src “kinst”
  https://cgit.freebsd.org/src/tree/sys/cddl/dev/kinst

- Tamches, Ariel & Miller, Barton P. “Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels”.

- The DWARF Standard. https://dwarfstd.org/

- Christos Margiolis “Using DWARF to find call sites of inline functions“. https://margiolis.net/w/dwarf_inline/

- Christos Margiolis “Inline function tracing with the kinst DTrace provider”. https://margiolis.net/w/kinst_inline/

- GitHub, inlinecall(1).
  https://github.com/christosmarg/inlinecall