sframes

Getting full user space stack trace
Acquiring user space stack traces

- Currently requires frame pointers
  - Perf also copies large amounts of the user space stack trace
- Frame pointers require setup at each function
  - More instructions to execute
- Frame pointers require a register to use
  - Pressure on register use
- Overhead: [https://lwn.net/Articles/919940/](https://lwn.net/Articles/919940/)
  - Kernel build: 2.4% increase
  - Blender test case: 2% increase
  - Python programs: Up to 10% increase!
Stack frames

- return addr
- previous fp
- variables

- fp
- sp
Stack frames

- return addr
- previous fp
- variables

- return addr
- previous fp
- variables

- fp

- sp
The Orc Unwinder

- Oops Rewind Capability
  - Needs something to go with DWARF and ELF formats!

- Much simpler than DWARF
  - Fewer and simpler annotations in the kernel asm files
  - Fewer bugs as a result (I know my DWARF annotations were very buggy)

- Added in 4.14 or live kernel patching
  - Live patching needed a reliable stack trace

- Created with tools/objtool at compile time

- Uses two tables
  - orc_unwind - orc_entry structures
  - orc_unwind_ip - IP addresses in the same offset as the orc_entry structure
Orc frames

```
func1:
[..]
```

```
return addr
variables
```

sp1

ip1
Orc frames

orc_unwind_ip:
[..]
ip1
[..]

return addr
variables

sp1

func1:
[..]

ip1
Orc frames

\[
\text{orc\_unwind\_ip:} \\
\text{[..]} \\
\text{ip1} \\
\text{[..]} \\
\text{entry1} \\
\text{sp\_offset = offset1} \\
\text{[..]}
\]
Orc frames

 orc_unwind_ip:  
[..]  
ip1  
[..]  
entry1

orc_unwind:

[..]

sp_offset = offset1

[..]

sp1 + offset1

sp1

func2:
[..]

ip2

func1:
[..]

ip1

return addr

variables
Orc frames

return addr

variables

sp1

func1:
[..]

func2:
[..]

orc_unwind_ip:
[..]
ip1
[..]
entry1
ip2
[..]

sp_offset = offset1

sp1 + offset1

sp1

orc_unwind:
[..]

ip2
Orc frames

```
func1:  [...]
func2:  [...]
```

```
sp1 + offset1
sp2
sp1
```

```
orc_unwind_ip:  [...]
orc_unwind:
entry1
sp_offset = offset1
```

```
return addr
return addr
variables
```

```
orc_unwind_ip:
ip1
orc_unwind:
entry1
sp_offset = offset1
ip2
```
Orc frames

sp1 + offset1

sp2

return addr variables

func2:
[..]

isp1

orc_unwind_ip:
[..]

ip1

[..]

entry1

sp_offset = offset1

ip2

[..]

entry2

sp_offset = offset2

orc_unwind:
[..]

func1:
[..]
Orc frames

return addr

variables

func1:

[..]

func2:

[..]

orc_unwind_ip:

[..]

ip1

entry1

ip2

entry2

sp_offset = offset1

sp_offset = offset2

sp1 + offset1

sp2 + offset2

sp2

sp2

sp1 + offset1

sp1

sp1

sp_offset = offset1

sp_offset = offset2

orc_unwind:

[..]
sframe

- sframe is based on orc, but for user space
- Can produce user space stack traces **without** frame pointers
- It is a section in the elf file with the two tables
- Requires to be compiled in and takes up disk space
- Can be read by the kernel
  - perf, ftrace and BPF can benefit from this
sframe in the kernel

- Can be done at the time of entering back into user space
  - The “ptrace” path
- Needs to handle offsets
  - Raw IP addresses are not helpful due to relocation
  - Can we convert them to the .text offset in the corresponding files?
- Perhaps even handle file names
  - Show not the address, but the `/proc/*/maps` info + offset into the file
Ptrace path

User space context  NMI  Kernel normal context

Interrupt context

NMI context

Perf wants stack trace

(Tries to read the user space stack)
Ptrace path

- User space context
- Kernel normal context
- Interrupt context
- NMI context

Perf wants stack trace
Asks sframe infrastructure for stack
Ptrace path

- User space context
- Kernel normal context
  - Is there work todo?
- Interrupt context
  - NMI context
  - Perf wants stack trace
  - Asks sframe infrastructure for stack
Ptrace path

User space context
Kernel normal context
Interrupt context
NMI context

NMI

Perf wants stack trace
Asks sframe infrastructure for stack

Is there work todo?
No
 PTRACE PATH

User space context

Kernel normal context

Interrupt context

NMI context

Perf wants stack trace

Asks sframe infrastructure for stack

Is there work todo?

Yes

Ptrace path
Ptrace path

User space context

Kernel normal context

Interrupt context

NMI context

NMI

Is there work todo?  Ptrace path

Perf wants stack trace

Asks sframe infrastructure for stack

Perf callchain
Ptrace path

User space context

Kernel normal context

Is there work todo?

Ptrace path

Interrupt context

NMI context

Perf wants stack trace

Asks sframe infrastructure for stack
Relocational addresses

Memory offset:
0x5a9fc3d08139

File offset:
0x1139

main:

0x1139

main() {

}
Use proc mapping

```
# cat /proc/3248/maps
555555554000-555555555000 r--p 00000000 fe:01 2157 /tmp/t
555555555000-555555556000 r-xp 00001000 fe:01 2157 /tmp/t
555555556000-555555557000 r--p 00002000 fe:01 2157 /tmp/t
555555557000-555555558000 r--p 00002000 fe:01 2157 /tmp/t
555555558000-555555559000 rw-p 00003000 fe:01 2157 /tmp/t
7fffffffdfc000-7fffffffdd2000 rw-p 00000000 00:00 0
7fffffffdd2000-7fffffffdd8000 r-xp 00000000 00:00 0
7fffffffdd8000-7fffffffdf2000 r--p 00000000 fe:01 313454 /usr/lib/x86_64-linux-gnu/libc.so.6
7fffffffdf2000-7fffffffdf7f000 r-xp 00000000 00:00 0
7fffffffdf7f000-7fffffffdfb3000 r--p 00000000 00:00 0
7fffffffdfb3000-7fffffffffd000 r-xp 00001000 00:00 0
7fffffffffd000-7ffffffffff000 r--p 00000000 00:00 0
7fffffffeff1000-7fffffffeff60000 r-xp 00000000 00:00 0
```

[vvar]
[vdso]
[stack]
[vsyscall]
SFrame for JITted code
Stack tracing for JITted code

- [WIP] Scoping
  - [#1] How does stack tracing work in JIT environments
  - [#2] Can SFrame make a difference
  - [#3] If yes, Identify the requirements for SFrame to support the JIT usecase

- Current understanding
  - [#1] Interpreted and Compiled application code, and VM’s own code
    - Runtime knows how to walk these variety of stack layouts
  - [#2] Potentially?
  - [#3] Let’s talk next...
SFrame for JIT code

● Pre-requisites
  ○ Stack layout of compiled/interpreted/VM code is psABI compliant.
    ■ Return address is either on stack (fixed location from CFA) or an ABI identified register

● JIT usecase
  ○ Functions may be added or removed
  ○ Functions size and content are modified over time
  ○ Functions may also be moved to a different location
  ○ Lifetime of JITted code varies

● Requirements
  ○ Growable SFrame section:
    ■ [At function granularity] Allow efficient addition of stacktrace data
    ■ [At function granularity] Allow efficient removal of stale stacktrace data
    ■ Support sorted and unsorted FDEs on PC for lookup
  ○ Lifetime-awareness: Allow efficient management of multiple .sframe sections
Growable SFrame section - I

- Support a “Growable” model
  - Create: Leave space in SFrame FDEs subsection, and SFrame FRE subsection
  - Append: If space available, add FDEs and the associated FREs; Else, create a new “Growable” SFrame section and copy over.
  - Delete: Mark as invalid. Provide means for deferred compaction.

- SFrame header flag
  SFRAME_F_FDE_SORTED
Growable SFrame section - II

- SFrame FDE: Add 1 bit to mark valid/invalid SFrame FDEs
  - Defer “Compaction” until when it’s a good time
- SFrame Header: Add new 32-bit offset to identify total size of SFrame FRE sub-section

Existing:

```c
uint32_t sfh_fre_len;
// Offset of SFrame Function Descriptor Entry section. */
```

- Add Compaction APIs for JIT Runtime
Lifetime-awareness using SFrame

- Lifetime of some jitted code is short
  - Address space gets reused
- Support multiple .sframe sections per process
  - Creation, management and bookkeeping may be left to the runtime
  - Is format level specification/support necessary?