Synthesized Call Frame Information for hand-written assembly in GNU assembler

Indu Bhagat
Linux Toolchains @ Oracle

Synthesizing Call Frame Information

- Current state of SCFI
- Does this help the current asm in the Linux kernel ?
- What other patterns should be accommodated to make this more useful ?

AS CFI directives

- From: User writes asm; Includes the necessary CFI annotations
- To: User write asm; Synthesize CFI in GAS

```
        .type   foo, @function
foo:
        .cfi_startproc                  ## CFA = rsp - 8
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16              ## CFA = rsp - 16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6         ## CFA = rbp - 16
# Begin %rsp manipulation for local stack usage (Dummy code)
        addq    %rax, %rdi
        movq    %rsp, %r12
        addq    $4, %rbx
        andq    $-16, %rax
        subq    %rax, %rsp
        movq    %rsp, %rdi
        call    bar
        movq    %r12, %rsp
# End %rsp manipulation for local stack usage
        mov     %rbp, %rsp
        .cfi_def_cfa_register 7         ## CFA = rsp - 16
        pop     %rbp
        .cfi_restore 6
        .cfi_def_cfa_offset 8           ## CFA = rsp - 8
        ret
        .cfi_endproc
```

SCFI mission statement

- Synthesize Call Frame Information[1] for assembly code
  - Hand-written asm (.S files)
  - Inline asm (asm () blocks)
- [1]Synthesize CFI rules for
  - **CFA and callee-saved registers**
    - **=> ABI / calling conventions are followed**

Can all CFI directives be synthesized for all input asm?

- TL;DR – No, but looks doable[1]
- Some directives indeed require user input
  - .cfi_signal_frame
  - .cfi_sections
  - .cfi_label
- [1] There are constraints that must be satisfied by input asm

New option --scfi[=all,none]

- Work in progress: --scfi=all (Aimed for hand-written asm)
  - Default, equivalent to --scfi
  - Ignores most CFI directives if present in input asm
    - Except .cfi_signal_frame, .cfi_label, .cfi_sections
- On the roadmap: For inline-asm, add new --scfi=inline
  - Does not ignore compiler generated CFI
  - Identifies #APP...#NO_APP and synthesizes CFI
- Also on roadmap: aarch64 support
- [binutils-gdb] [[PATCH, V2 00/10] Synthesize CFI for hand-written asm] https://sourceware.org/pipermail/binutils/2023-October/130210.html

Eligibility Criteria, a.k.a., “Constraints” for hand-written (non-inline) asm

Discuss: How much does each constraint limit practical usages of asm in the Linux kernel ?

Trailer...

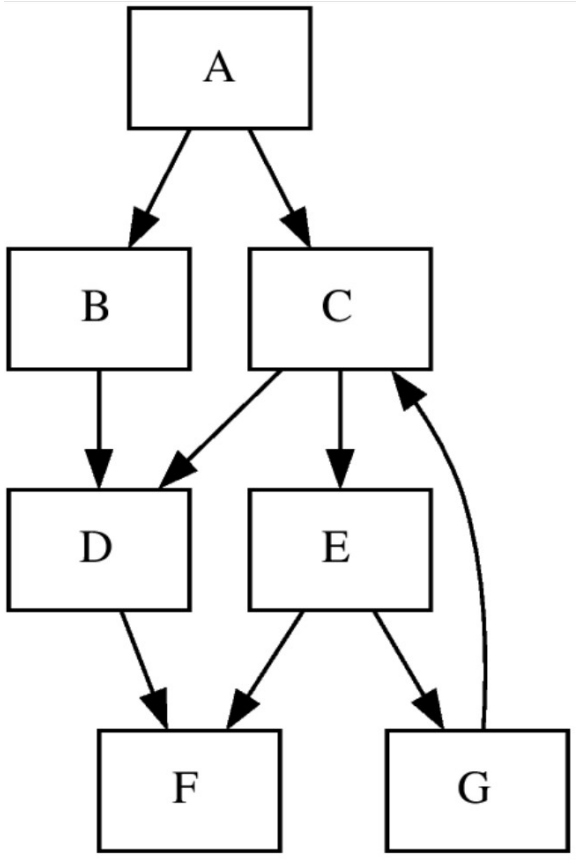
- ABI/calling convention conformant code
- Amenable to asynchronous stack unwinding
- CFA must be REG_SP or REG_FP based
- CFA base register must be traceable at all times
- Code with indirect branches, jump tables not supported

(#1) Identifying beginning and end of code block

- **Must begin with**
  - **.type name, @function ## beginning of func**
- Closing with .size name, .-name ## end of func
  - **Recommended** if single section
  - **Necessary** if interleaving text sections (e.g., when using .section .text.unlikely / .section .rodata / .pushsection / .popsection etc.)
- PS: Not applicable for inline asm (#APP...#NO_APP)

(#2) Deciphering the control flow unambiguously

- Issue: It is not possible to reconstruct the complete control flow graph from assembly
  - Indirect jumps, jump table
- Warning: Untraceable control flow for func ‘foo’. Skipping SCFI.



Input asm follows some conventions

- ABI/calling conventions
  - (#2) Symmetric save and restore
    - Warning: SCFI: asymetrical register restore
  - (#3) Balanced stack at return
    - Detection and Warning TBD
- Amenable to asynchronous stack ~~tracing~~ unwinding
  - (#4) Code must not clobber the base register used for CFA tracking in an untraceable way

(#4) Base register for CFA must be traceable at all times

- DWARF5 says CFA: [reg + offset], ~~or DWARF expression~~
- Static stack allocation:
  - (#4a) Stack location (REG_SP) is traceable at each save (push) and restore (pop) of callee-saved registers
- Dynamic stack allocation:
  - (#4b), (#5) next...

(#4b) Switch to reg FP for dyn stack alloc

```
foo:
        .cfi_startproc
        pushq       %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
# Begin %rsp manipulation for local
        addq    %rax, %rdi
        movq    %rsp, %r12
        addq    $4, %rbx
        andq    $-16, %rax
        subq    %rax, %rsp
        movq    %rsp, %rdi
        call    bar
        movq    %r12, %rsp
# End %rsp manipulation for local s
        mov         %rbp, %rsp
        .cfi_def_cfa_register 7
        pop         %rbp
        .cfi_restore 6
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

- DRAP usage is not supported, but can be accommodated.
- **Switch to any other callee-saved register is NOT supported.**

(#5) CFA base register must be REG_SP or REG_BP

In Summary, SCFI has some eligibility criteria...

- ABI/calling convention conformant code
- Amenable to asynchronous stack unwinding
- CFA must be REG_SP or REG_FP based
- CFA base register must be traceable at all times
- Code with indirect branches, jump tables not supported

Discuss

- How much does this limit practical usages of asm in the Linux kernel ?
- What other patterns should be accommodated to make this more useful ?
- How is the stack trace info of the alternatives currently being updated when executable code is patched ?

Extra

```
        .type   foo, @function
foo:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $17, %esi
        movl    $5, %edi
        call    add
        .section        .rodata
        .align 16
        .type   __test_obj.0, @object
        .size   __test_obj.0, 24
__test_obj.0:
        .string "test_elf_objs_in_rodata"
.LC0:
        .string "the result is = %d\n"
        .text
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa_register 7
        .cfi_restore 6
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
```

```
# Testcase where a user may define hot and
# cold areas of function
        .globl  foo
        .type   foo, @function
foo:
        .cfi_startproc
        testl   %edi, %edi
        je      .L3
        movl    b(%rip), %eax
        .section        .text.unlikely
        .type   foo.cold, @function
        .cfi_startproc
foo.cold:
.L3:
        pushq   %rax
        .cfi_def_cfa_offset 16
        call    abort
        .cfi_endproc
.LFE11:
        .text
        ret
        .cfi_endproc
        .size   foo, .-foo
        .section        .text.unlikely
        .size   foo.cold, .-foo.cold
```

.type and .size are needed to make boundaries unambiguous when section interleaving

```
        leaq    8(%rsp), %r10
        .cfi_def_cfa 10, 0
        andq    $-16, %rsp
        movl    $1, %edi
        pushq   -8(%r10)
        pushq   %rbp
        movq    %rsp, %rbp
        .cfi_escape 0x10,0x6,0x2,0x76,0
        pushq   %r10
        .cfi_escape 0xf,0x3,0x76,0x78,0x6
        subq    $24, %rsp
        call    _Z3bari
        movl    i(%rip), %edx
        movl    %edx, %eax
        testl   %edx, %edx
        jne     .L4
.L19:
        testl   %eax, %eax
        jne     .L14
        movq    -8(%rbp), %r10
        .cfi_def_cfa 10, 0
        leave
```

```
        .globl  self_aligning_foo
        .type   self_aligning_foo, @function
self_aligning_foo:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
# The following 'and' op aligns the stack pointer.
        andq    $-16, %rsp
        subq    $32, %rsp
        movl    %edi, 12(%rsp)
        movl    %esi, 8(%rsp)
        movl    $0, %eax
        call    vector_using_function
        movaps  %xmm0, 16(%rsp)
        movl    12(%rsp), %edx
        movl    8(%rsp), %eax
        addl    %edx, %eax
        leave
# GCC typically generates a '.cfi_def_cfa 7, 8' for leave
# insn.  The SCFI however, will generate the following:
        .cfi_def_cfa_register 7
        .cfi_restore 6
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   self_aligning_foo, .-self_aligning_foo
```