Synthesizing Call Frame Information for hand-written assembly

Bhagat, Indu
indu.bhagat@oracle.com

September 20, 2023
V1

Abstract

Call Frame Information (or CFI) is the information that accompanies executable ELF object code and is used to perform virtual stack unwinding or stack walking at any point of the execution of that code. There are several formats available to encode CFI data in ELF files; DWARF Frame, EH Frame and SFrame are some of them.

Compilers are able to generate the CFI annotations corresponding to the code they compile, completely and accurately. However, there are scenarios where assembly language programs are written and maintained by hand; these programs can, occasionally, be large. In hand-written assembly language programs, the CFI pseudo-ops are either added by hand or, what is more usual, completely absent. Annotating assembly language programs with CFI directives is an error-prone task, and maintaining them as the code evolves is time consuming. Some applications like the Linux kernel avoid the problem of maintaining CFI annotations by using their own in-house tools to post-process compiled object codes: the call frame information (for stack tracing) is reverse-engineered from the binary. These reverse-engineering solutions use knowledge about the ISA and the ABI to infer the stack frame structure from the instructions. Although such a solution involving reverse-engineering of binaries works in practice[1] for at least one architecture, it is neither scalable nor easy to maintain.

In this paper, we propose to add support to the GNU assembler so that it can synthesize CFI information for the programs it assembles, even in the absence of CFI directives in the assembly sources. We have started to implement this functionality in a prototype and it is already looking very promising. A very central concept of this proposal and implementation is that the GNU assembler now ”understands” some of the machine instructions it assembles. Before, the common assembler code would only see fragments provided by the backends; now, the backends can optionally provide additional information in the form of generic instructions; this new infrastructure can be used as a foundation of other interesting features in the assembler, such as certain optimizations, code property validation and program verification.

1 Introduction to DWARF CFI

The DWARF Call Frame Information, defined in the DWARF debugging standard [1], is typically used to convey stack unwind information, per PC, for the generated programs. This information is used by debuggers, profilers, and many other program analysis tools to generate backtraces and also state recovery.

[1]does have issues deciphering some control flow

```
0000921c 0000000000000020 00009220 FDE cie=00000000 pc=000000000046451a..0000000000464559
  DW_CFA_advance_loc: 5 to 000000000046451f
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r6 (rbp) at cfa-16
  DW_CFA_advance_loc: 3 to 0000000000464522
  DW_CFA_def_cfa_register: r6 (rbp)
  DW_CFA_advance_loc: 3 to 0000000000464525
  DW_CFA_offset: r12 (r12) at cfa-24
  DW_CFA_offset: r3 (rbx) at cfa-32
  DW_CFA_advance_loc: 51 to 0000000000464558
  DW_CFA_def_cfa: r7 (rsp) ofs 8
  DW_CFA_nop
  DW_CFA_nop
```

Figure 1: DWARF opcodes in an FDE in a sample `.eh_frame` section

```
0000921c 0000000000000020 00009220 FDE cie=00000000 pc=000000000046451a..0000000000464559
   LOC           CFA      rbx   rbp   r12   ra
000000000046451a rsp+8    u     u     u     c-8
000000000046451f rsp+16   u     c-16  u     c-8
0000000000464522 rbp+16   u     c-16  u     c-8
0000000000464525 rbp+16   c-32  c-16  c-24  c-8
0000000000464558 rsp+8    c-32  c-16  c-24  c-8
```

Figure 2: Interpreted DWARF CFI from the FDE in Figure 1

Conceptually, DWARF CFI specifies how to recover the return address and callee-saved registers at each PC in a given function. When available, the DWARF CFI information can be found in dedicated sections (e.g. `.debug_frame` or `.eh_frame`) in the object files.

Figure 1 shows an excerpt from an `.eh_frame` section. The excerpt corresponds to the DWARF CFI information for one function in the binary. A simple `objdump -Wf`[2] on your favorite binary will show the dump of the DWARF CFI contained in the `.eh_frame` section.

The `DW_CFA_*` opcodes, in Figure 1, are the DWARF CFI opcodes which when executed, help the unwinder or stack tracer recover the required register: `CFA` (Canonical Frame Address), `ra` (Return Address), or any of the callee-saved registers. In aggregate (interpreted) form, the above bytecodes generate information equivalent to the following (dump generated using `objdump -WF`), as shown in Figure 2. For deeper understanding of the DWARF CFI format, the readers are encouraged to peruse the DWARF debugging standard specification[1].

The GNU assembler defines a set of CFI directives [2], which are used by the compiler to convey the call frame information. Figure 3 shows a simple example with x86_64 assembly and its associated GNU AS CFI directives. With respect to support in the GNU toolchain, `gcc` is capable of generating accurate CFI annotations[3] for all code written in high-level language. The GNU assembler consumes these CFI annotations to then generate stack unwind information in the format chosen by the user (EH Frame, Debug Frame etc.).

These CFI pseudo-ops are also used by assembly-code programmers to convey stack unwind information for their hand-written asm code. Manually adding CFI annotations to assembly programs needs additional expertise; human-errors are possible and indeed occur more often than one may like. These errors, if present at the time of virtual stack unwind, lead to unfavorable outcomes: incorrect stacktraces, program state corruption or even a crash at an inopportune

[2]On x86_64

[3]The keyword 'CFI annotations' is used interchangeably in this paper to imply 'GNU AS CFI directives'

```
        .text
        .globl  foo
        .type   foo, @function
foo:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        [...]
        popq    %rbp
        .cfi_def_cfa_offset 8
        .cfi_restore 6
        ret
        .cfi_endproc
.LFE0:
        .size   foo, .-foo
```

Figure 3: Using GNU AS CFI directives on x86_64

time.

The motivation behind this work is to enhance the capabilities of the GNU Toolchain so that it can synthesize CFI for hand-written assembly and hence, alleviate the user of the task of writing CFI annotations by hand. We refer to this capability of Synthesizing CFI as **SCFI** in the rest of the document.

2 Synthesis of CFI (SCFI) for assembly

To design a solution for synthesis of CFI for assembly, one of the key question that needs to be resolved is:

`Q1`: Is it possible to synthesize CFI for any arbitrary asm out there?

For this particular question, despite not being exactly the same, it is useful to think about the synthesis of CFI as the synthesis of the corresponding CFI directives, simply for sake of understanding. In other words, the question is:

Given an assembly program, if the task is to "complete" it with CFI pseudo-ops whose evaluation would result in correct CFI for the corresponding code, can we generate all the necessary CFI psuedo-ops that the user may have had to manually specify ?

Now, in essence, to automatically synthesize CFI, we have the following two sources of information available to us:

1. The assembly code
2. Knowledge of the ABI and calling convention

In order to answer this question, first lets note that there indeed exist some `.cfi_*` directives which *cannot* be synthesized by the assembler using only the information available in these sources. This is because the above two sources of information are not sufficient in some cases.

2.1 Non-synthesizable GAS CFI directives

According to our understanding as of today, a complete list of CFI directives that we cannot synthesize by looking at the hand-written asm are as follows:

1. `.cfi_signal_frame` : This directive marks the current function as signal trampoline.

2. `.cfi_sections`: This directive is used to specify which section(s) should the CFI information be emitted to: `.eh_frame` section, `.debug_frame` section and/or `.sframe` section, etc.

   The default in the GNU assembler is `'.cfi_sections .eh_frame'` (if no `.cfi_sections` is specified). Further, for generating the `.sframe` section, GAS also supports a command line option `--gsframe`. Adding new command line options, like that for `.sframe` may provide a way to get user input, but it needs some careful thought: A user may want to specify a list of sections, like `'.cfi_sections .eh_frame, .debug_frame'`. Or, the user may choose `'.cfi_sections .eh_frame_entry'` which is to generate Compact EH Frame. Adding command line options does not seem scalable here.

3. `.cfi_label`: This CFI directive allows to identify CFI data explicitly. This allows alteration of stack unwind data, if needed, when the original code sequence in the function needs to be patched for some reason. See the original commit for more information [4]. `glibc` has some uses of `.cfi_label`, e.g., in sysdeps/arc/start.S and sysdeps/unix/sysv/linux/riscv/clone.S.

4. `Others`: There are other CFI directives which also fall in the category of non-synthesizable CFI directives. These include `.cfi_personality`, `.cfi_personality_id`, `.cfi_lsda` and `.cfi_inline_lsda`. Although many of these directives are used for Compact EH Frame format, more careful thought is needed to holistically continue to support the existing GAS functionality.

PS: On the `aarch64` side, there are special CFI directives for managing code using Pointer Authentication. E.g., `.cfi_b_key_frame`[3]. The latter *should* be synthesizable by looking at whether it is the `paciasp`, or `pacibsp` instruction.

Current Status: The CFI directives mentioned in the above-list cannot be generated by the SCFI machinery as they need user-input. This has implications on the overall offering: note that not handling the user-specified `.cfi_sections` implies that the proposed implementation will emit CFI to the default section - `.eh_frame`. Similarly, the other implications are easy to extrapolate.

Adding a command line with good defaults and the appropriate arguments may help manage some of the above-mentioned directives, but clearly not all. Some of the directives apply at a per-function granularity, so a command line option may not work out.

Summary: In summary, there indeed are some CFI directives which cannot be auto-generated; the user is the only entity that can specify them. In the current proposal (and it's design and implementation), these CFI directives have been set aside for now. They need to be accommodated after some more discussion.

2.2 High-level goal of the proposal

With that said, it is important now to define the goal at this point:

Generating CFI for all compiler-generated asm is NOT the goal;
Generating CFI for most patterns found in practice in hand-written asm is the goal.

When it comes to code generation, a compiler has more context than any other component in the toolchain. The compiler may generate a variety of complex code patterns together with the accompanying CFI information accurately as it has the full context of the user-defined functions. Some compiler-generated asm, e.g., the usage of Dynamic Realigned Argument Pointer (DRAP) to realign stack may be tricky to comprehend for the assembler. There are other code patterns like indirect jumps, jump tables, etc. (more on this in later sections) which cause difficulty for

SCFI. Hence, unless there is a requirement to handle compiler-generated code, it appears both non-trivial and not very practical to automatically generate CFI for compiler-generated code.

In this proposal, we focus on synthesizing CFI for most hand-written asm. As we present the code examples in further sections, one may see that indeed there are some ”restrictions” [4] around the hand-written assembly so that GNU assembler can synthesize CFI.

2.3 New comand line option: `--scfi[=all,none]`

We propose to add a new command line option `--scfi[=all,none]` to the GNU assembler. The default argument is `all`. Hence, `--scfi` or `--scfi=all` instructs the assembler to synthesize CFI. The CFI is then emitted in the sections corresponding to whatever format are specified when calling the assembler: `.eh_frame` (default), `.sframe` (when invoked with `--gsframe`) etc.

Moving forward, the assembler may want to support three operation modes with respect CFI synthesis:

1. `--scfi=none`: Do not synthesize any CFI.
2. `--scfi=all`: Synthesize CFI info for the whole assembly unit (i.e. the .s file). All user-provided CFI directives are ignored by GAS with this command line option [5].
3. `--scfi=inline`: Synthesize CFI info only for inline assembly generated by the compiler.

DISCUSS: We should support a distinct option of `--scfi=inline` for inline asm, where the GNU assembler consumes (and not ignores) the compiler generated CFI for the code surrounding the inline asm. In that mode, the SCFI implementation should process these compiler-generated CFI directives (instead of synthesizing them) in order to establish the right CFI unwind state before the inline asm block.

Note that the assembler should be able to determine which parts of the assembly code is inlined by looking for comments generated by the compilers:

```
#APP
[...]
#NO\_APP
```

2.4 Background

In this section, we will take a look at some assembly language sequences along with their respective accompanying CFI assembler directives. This should be helpful to then lay the background necessary to understand what is needed for synthesizing CFI automatically. Most of the assembly code stubs shown in this document use `x86_64` insns, but the concepts should be applicable equally well to other popular ISAs.

2.4.1 Knowing the stack usage

Clearly, knowing the stack usage is *essential* for knowing the location of the register saves and for possibly validating the register restores [6]. Walking through Figure 4 will help understanding this requirement.

Note how in Figure 4, the CFA at all times is `REG_SP`-based. This means, the stack size used by the function must be known at *all times* to synthesize CFA location. The example is what **static stack usage** pattern looks like.

[4] mostly practical restrictions. We aim to make the implementation as programmer-friendly as possible.

[5] This needs more discussion as we still need a way to deal with those `.cfi_*` directives which cannot be auto-generated. See section 2.1.

[6] of the callee-saved registers

Figure 4: Example to showcase CFI for callee-saved registers and static stack usage

```
        .text
        .globl        foo
        .type        foo, @function
foo:
        .cfi_startproc                  ## CFA = rsp -8
        pushq        %r12
        .cfi_def_cfa_offset 16          ## CFA = rsp -16
        .cfi_offset 12, -16
        pushq        %r13
        .cfi_def_cfa_offset 24          ## CFA = rsp - 24
        .cfi_offset 13, -24
# The function may use callee-saved registers for its use, and may even
# choose to spill them to stack if necessary.
        addq    %rax, %r13
        subq         $8, %r13
# These two pushq's of callee-saved regs must NOT generate
# .cfi_offset.
        pushq        %r13
        .cfi_def_cfa_offset 32         ## CFA = rsp - 32
        pushq        %rax
        .cfi_def_cfa_offset 40         ## CFA = rsp = 40
# Function manipulates %rsp to get rid of local stack usage.
        addq        $16, %rsp
        .cfi_def_cfa_offset 24        ## CFA = rsp - 24
# The SCFI machinery must keep track of where the callee-saved registers
# are on the stack.  It should generate a restore operation if the stack
# offsets match.
        popq        %r13
        .cfi_restore 13
        .cfi_def_cfa_offset 16        ## CFA = rsp - 16
        popq        %r12
        .cfi_restore 12
        .cfi_def_cfa_offset 8        ## CFA = rsp - 8
        ret
        .cfi_endproc
.LFE0:
        .size        foo, .-foo
```

Figure 5: Dynamic stack usage using frame-pointer register for tracking CFA

```
# Example to showcase switching between sp/fp based CFA.
        .text
        .globl        foo
        .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
.LFE0:
        .size        foo, .-foo
```

Of course, some functions will need to perform **dynamic stack allocation**. In the case of programs written in a high-level language, the compiler will systematically fall back on the usage of frame-pointer register for tracking the CFA. For the purpose of synthesizing the CFI, we assume that asm programmers will also follow a similar style. Please see Figure 5 to help understand this code pattern.

2.4.2 Identifying the `CFA`

As shown in the example in Figure 4, when the function performs static stack allocation and when the `CFA` is defined as an offset to the `REG_SP`, it is important that `REG_SP`, a.k.a., the stack usage is precisely known at all times. Similarly, for the case of dynamic stack usage, at all times where the function uses `REG_FP` based `CFA`, the precise changes to the value of the `REG_FP` must be known.

In other words, the precise changes to the register used as the base for identifying the `CFA`, referred to as the `CFA base register` in the rest of the document, must be known. Now, the DWARF standard[1] defines the `CFA` as follows:

"The CFA column defines the rule which computes the Canonical Frame Address value; it may be either a register and a signed offset that are added together, or a DWARF expression that is evaluated."

Hence, for compiler-generated code, the CFA can even be a DWARF expression. For the pupose of SCFI, however, it makes sense to limit the `CFA` tracking to be register based only.

Hence,

Rule 1: At all times in a function, the CFA value must be a register and a signed offset that are added together.

Further, whether it is static or dynamic stack usage, it makes sense to limit the base-register for CFA tracking to two registers, without limiting the usefulness of the proposal. Hence, for the purpose of synthesizing CFI,

Rule 2: The SCFI machinery in GAS will only allow for two `CFA` base registers:

1. *The stack-pointer register (REG_SP) itself, and*
2. *The frame-pointer register (REG_FP)*

This implies that those functions using dynamic realigned argument pointer (DRAP) to realign stack are not currently supported. More on DRAP and synthesizing CFI for functions which dynamically realign the stack later. That said, including the basic support for DRAP should be doable while adhering to the above-mentioned rules for the rest of the assembly code.

2.4.3 Control-flow matters

Some functions may have multiple points of return. Each point of return may have its distinct epilogue. To correctly synthesize CFI for such asm functions, it is important to take into account the control flow inside the function.

PS: The complication here is that it *may not* be possible to generate a precise control flow graph from assembly. There is no way for us to know the branch targets of indirect jumps.

Figure 6 shows an example of a function with two possible return paths. The compiler generates a set of `.cfi_remember_state` and `.cfi_restore_state` to tackle the problem of synthesizing CFI opcodes in wake of change of flow instructions, especially conditional branches. The SCFI machinery must perform similarly to ensure correctness.

3 Design and Implementation

In this section, we will go over some key aspects of the design and implementation of the SCFI feature in GNU assembler.

3.1 Identifying the boundaries of asm functions

For synthesizing CFI, the GAS will firstly need to identify the beginning and end of an assembly function. Stack unwind information follows closely the assembly code of the function and hence, by its nature, starts anew for each function. Luckily, the existing GNU AS directives come to the rescue.

The implementation in the GNU assembler relies on seeing the following two markers to identify the beginning and end of an assembly function.

For generating SCFI, a function must begin with the following pseudo-op:

```
.type <func-name>, @function
```

Lastly, the function must end with the following pseudo-op:

```
.size <func-name>, -.<func-name>
```

Figure 6: Control flow matters. Function with two return paths

```
        .globl        main
        .type        main, @function
main:
        .cfi_startproc            ## CFA = rsp - 8
        pushq        %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16        ## CFA = rsp - 16
        movl        $.LC0, %esi
        movl        $.LC1, %edi
        call        fopen
# Two return paths for
# return ferror (f) || fclose (f) != 0;
        movq        %rax, %rdi
        movq        %rax, %rbx
        call        ferror
        movl        %eax, %edx
        testl        %edx, %edx
        je        .L7
# Exit BB 1
        popq        %rbx                  ## CFA = rsp - 8
        .cfi_remember_state
        .cfi_def_cfa_offset 8
        .cfi_restore 3
        ret
.L7:
# Exit BB 2
        .cfi_restore_state           ## CFA = rsp - 16
        movq        %rbx, %rdi
        call        fclose
        popq        %rbx                  ## CFA = rsp - 8
        .cfi_def_cfa_offset 8
        .cfi_restore 3
        testl        %eax, %eax
        setne        %al
        movzbl        %al, %eax
        ret
        .cfi_endproc
        .size        main, .-main
```

3.2 The GAS instruction: `ginsn`

We define a data structure called the **GAS instruction**, a.k.a., the `ginsn`. This the fundamental token of information exchange about executable code from the targets to the target-independent component of the GNU assembler.

The definition of `struct ginsn` is presented below, and is taken from the `ginsn.h` header file in gas source code. A `ginsn`, may need further enhancements when other targets choose to use the SCFI machinery. At this time, a `ginsn` has two source operands and one destination operand. Further details should be easy to find in the aforementioned header file.

As noted in the code comments, there may be more than one `ginsn` per machine instruction. This may be true for both `RISC` or `CISC` ISAs.

At the moment, the `ginsn` format of instruction representation is lossy: Not all information from the target instruction is encoded into the `ginsn`. Not all target instructions need to be translated into the corresponding set of `ginsn`[7]. As and when more use-cases for `ginsn` are found, this status quo will shift.

```
/* GAS generic instruction.

   Generic instructions are used by GAS to abstract out the binary machine
   instructions.  In other words, ginsn is a target/ABI independent internal
   representation for GAS.  Note that, depending on the target, there may be
   more than one ginsn per binary machine instruction.

   ginsns can be used by GAS to perform validations, or even generate
   additional information like, sythesizing CFI for hand-written asm.

   FIXME - what back references should we keep - frag ? frchainS ?
   */

struct ginsn
{
  enum ginsn_type type;
  /* GAS instructions are simple instructions with GINSN_NUM_SRC_OPNDS number
     of source operands and one destination operand at this time.  */
  struct ginsn_src src[GINSN_NUM_SRC_OPNDS];
  struct ginsn_dst dst;
  /* Additional information per instruction.  */
  uint32_t flags;
  /* Symbol.  For ginsn of type other than GINSN_TYPE_SYMBOL, this identifies
     the end of the corresponding machine instruction in the .text segment.
     These symbols are created anew by the targets and are not used elsewhere
     in GAS.  These can be safely cleaned up when a ginsn is free'd.  */
  symbolS *sym;
  /* Location information for user-interfacing messaging.  */
  const char *file;
  unsigned int line;

  /* Information needed for synthesizing CFI.  */
  scfi_opS **scfi_ops;
  uint32_t num_scfi_ops;
```

[7]Fundamentally speaking, only the information strictly necessary for correctness of SCFI is brought the GAS using the `ginsn` abstraction.

```
  /* Flag to keep track of visited instructions for CFG creation.  */
  bool visited;

  ginsnS *next; /* A linked list.  */
};
```

3.3 Target-specific functionality to generate `ginsn`

As noted earlier, for targets choosing SCFI, translating all machine instructions into their corresponding `ginsn` will be wasteful and above all, unnecessary.

So the question then comes:

Q2: What are the set of target instructions that are necessary to ensure the SCFI machinery generates correct CFI for the hand-written asm?

To the best of our understanding so far, the following set of instructions are critical for ensuring synthesis of correct CFI[8]:

1. All change of flow instructions, including all conditional and unconditional branches, call and return from functions.
2. All register saves and unsaves to the stack.
3. All instructions affecting the two registers that could potentially be used as the base register for CFA tracking. Recall that we have limited the base-register for CFA tracking to `REG_SP` and `REG_FP`.

Q3: What is the right time, in the GAS workflow, to translate a target instruction into it's corresponding set of `ginsn` ?

Ideally, this should be done when the target instruction is a) fully known, and b) done with all the intended optimizations. Further, this process must be undertaken for each target instruction included in the answer to *Q2* above. This leaves us to choose some point after the target has successfully called `output_insn ();` in the backend.

3.4 Fragments and fix-ups

The GAS abstraction of `fragment` forms the back-bone of much of the inner workings of GAS. It is a concept which is used to identify a blob of data which eventually makes into an output section. An important aspect of a `fragment`'s life-cycle is an operation called "fix-up".

Unfortunately, as for the second requirement of "be done with all the intended optimizations", it is not always possible to guarantee that all the intended optimizations are done as some "fix-ups" may be performed by the backend later in time. This remains an open issue, but one that, we think, should not impact the correctness of SCFI: Even if a backend may optimize the instruction patterns, the semantics of the target instruction will remain the same.

3.5 Algorithm for SCFI

To bring it all together, the following figure outlines the algorithm implemented in GNU assembler to synthesize CFI.

synth_dw2cfi (): This function takes as an input the list of `ginsn`(s) generated per function. The GAS target generates this list of `ginsn`(s), as it assmebles each machine instruction.

[8] also verified via the SCFI implementation for `x86_64`

To generate CFI, at least three passes per function are needed. The first two passes ares over the list of `ginsn`, whereas the third pass is over the set of basic blocks in a the `gcfg`, the control flow graph.

Here is the workflow with the three steps:

1. First, to create control flow graph via **build_cfg ()**,
2. Second, forward pass to propagate the SCFI state per basic block (BB) via **forward_flow_scfi_state ()**,
3. Third, backward pass to generate any `.cfi_remember_state` and `.cfi_restore_state`, if necessary. This is done via **backward_flow_scfi_state ()**.

forward_flow_scfi_state (): This function is the backbone of CFI generation as the bulk of the CFI are generated here. This function also implements the rules specified earlier (See *Rule 1* and *Rule 2* in section 2.4.2). Each `ginsn` is processed and the SCFI unwind state object is updated. A snapshot of the SCFI unwind state object is kept at the entry and exit of each basic block for both functionality and implementing correctness checks (See how **cmp_scfi_state ()** is called when we land at already visited basic blocks)

backward_flow_scfi_state ()[9]**:** This function processes the set of basic blocks in reverse PC order. With this backward traversal, one can find points of return from the function with differing SCFI unwind states.

```
int synth_dw2cfi (ginsnS *ginsn):
  gcfg = build_cfg (ginsn);
  root_bb = get_rootbb_gcfg (gcfg);
  init_cfi_state = scfi_state_new ();

  /* Traverse the cfg and update the scfi_op per ginsn.  */
  ret = forward_flow_scfi_state (gcfg, entry_bb, init_state);
  if (ret) handle_bad ();
  /* Traverse the cfg in reverse IP order and generate .cfi_restore_state
     and cfi_remember_state as necessary.  */
  ret = backward_flow_scfi_state (gcfg);
  if (ret) handle_bad ();
  return ret;
```

```
/* Recursively propagate STATE starting at BB of GCFG.  */
int forward_flow_scfi_state (gcfg, bb, state):
  if (bb->visited)
    /* Check that the CFI state is the same as seen previously when landing
       from another branch.  */
       ret = cmp_scfi_state (state, bb->entry_state);
       if (ret)
           handle_bad ();
           return ret;

  /* Initialize the BB's SCFI state at entry with the given STATE object.  */
  gbb->entry_state = scfi_state_init (state);

  /* Perform symbolic execution of each ginsn in the gbb and update the
     scfi_ops list of each ginsn.  */
  bb_for_each_insn(gbb, ginsn)
      ret = gen_scfi_ops (ginsn, state);
      if (ret) goto fail;

  /* Initialize the BB's SCFI state at exit with the updated STATE object.  */
```

[9]TBD: This routine needs more eyes and testing. It works for generating a single set of remember/restore CFI, but there can be nested remember/restore

```
  gbb->exit_state = scfi_state_init (state);
  gbb->visited = true;
  /* Process the next BB in DFS order.  */
  prev_bb = gbb;
  if (gbb->num_out_gedges)
      bb_for_each_edge(gbb, gedge)
          gbb = gedge->dst_bb;
          /* For a BB already visited, the scfi_state at entry of BB must match
             the current STATE.  */
          if (gbb->visited && cmp_scfi_state (gbb->entry_state, state))
              goto fail;

          if (!gedge->visited)
              gedge->visited = true;

              memcpy (state, prev_bb->exit_state, sizeof (scfi_stateS));
              ret = forward_flow_scfi_state (gcfg, gbb, state);
              if (ret) goto fail;
  return 0;
fail:
  gedge->visited = true;

  return 1;
```

```
int backward_flow_scfi_state (gcfg):
 gcfg_get_bbs_in_prog_order (gcfg, prog_order_bbs);

  i = gcfg->num_gbbs - 1;
  /* Traverse in reverse program order.  */
  while (i > 0)
      current_bb = prog_order_bbs[i];
      prev_bb = prog_order_bbs[i-1];
      if (cmp_scfi_state (prev_bb->exit_state, current_bb->entry_state))
          /* Candidate for .cfi_restore_state found.  */
          ginsn = bb_get_first_ginsn (current_bb);
          scfi_op_add_cfi_restore_state (ginsn);
          /* Memorize current_bb now to find location for its remember state
             later.  */
          restore_bbs[i] = current_bb;
      else
          bb_for_each_edge (current_bb, gedge)
              dst_bb = gedge->dst_bb;
              for (j = 0; j < gcfg->num_gbbs; j++)
                if (restore_bbs[j] == dst_bb)
                    ginsn = bb_get_last_ginsn (current_bb);
                    scfi_op_add_cfi_remember_state (ginsn);
                    /* Remove the memorised restore_bb from the list.  */
                    restore_bbs[j] = NULL;
                    break;
      i--;

  /* All .cfi_restore_state pseudo-ops must have a corresponding
     .cfi_remember_state by now.  */
  ret = check_restore_bbs_is_null_p (restore_bbs);

  return ret;
```

3.6 SCFI warnings and errors in GAS

A set of warnings and errors have been added to the current SCFI implementation. Every situation where GAS is not sure it will be able to synthesize valid CFI is treated as an error[10]. Following is a subset of the warnings and errors:

1. Warning: SCFI: asymetrical register restore
2. Error: SCFI: usage of REG_FP as scratch not supported
3. Error: SCFI: unsupported stack manipulation pattern

4 Other Use-cases

The addition of the infrastructure for creation of `ginsn`, the control flow graph of `ginsn`, opens up the window to implement other useful features in GAS, apart from SCFI.

4.1 Validation of compiler generated CFI

One may wonder if this infrastructure can be used to automatically validate compiler generated CFI. Recall that for correct SCFI, control flow matters (see section 2.4.3). Code stubs with indirect jumps, jump table etc. are expected for compiler generated assembly. Further, there are practical restrictions around what assembly code is ingestible for SCFI (see *Rule 1* and *Rule 2*) in section 2.4.2).

Even for cases when above-mentioned issues are non-existent, there, unfortunately, remain some practical differences between the CFI generated by compiler vs. those synthesized by the GNU assembler's SCFI machinery. A naive comparison of the two is not possible; or at least such a task may be harder than desired.

The following table shows a subset of cases where differences are seen.

GCC generated	GNU AS generated	Comments

`popq %rbx`
`.cfi_def_cfa_offset 8`
`.cfi_restore 3`
	`popq %rbx`
	`.cfi_restore 3`
	`.cfi_def_cfa_offset 8`
		Note the order of the two CFI pseudo-ops.

`testl %edx, %edx`
`je .L7`
`popq %rbx`
`.cfi_remember_state`
`.cfi_def_cfa_offset 8`
`ret`
`.L7:`
`.cfi_restore_state`
`movq %rbx, %rdi`
`call fclose`
	`testl %edx, %edx`
	`je .L7`
	`.cfi_remember_state`
	`popq %rbx`
	`.cfi_restore 3`
	`.cfi_def_cfa_offset 8`
	`ret`
	`.L7:`
	`.cfi_restore_state`
	`movq %rbx, %rdi`
	`call fclose`
		Sometimes, the compiler does not generate `.cfi_restore` for insns in epilogue. Secondly, GNU AS generates generates the remember and restore state ops at different instructions.

[10]as CFI synthesis is explicitly requested by the user via the command line

4.2 Further validation of hand-written asm

The SCFI machinery is consumer of the set of `ginsn`. In that sense, `ginsn` provide GAS with an abstraction to understand the instructions generated by the targets. Using `ginsn`, GAS is now capable of sufficient context at each asm instruction. As noted in section 3.6, GAS can already issue some useful messaging to the user, in the context of SCFI.

Using the `ginsn` abstraction, it is possible to make GAS emit other useful diagnostics like[11]:

1. Missing CFI save for CFI restore
2. Missing CFI restore for CFI save
3. Missing user-defined label for conditional branch
4. Unreachable code

As you see, some of these warnings, especially the last two, are not SCFI related. These, and more, are other use-cases of supporting the `ginsn` abstraction.

4.3 Limited program verification for BPF

Writing BPF programs so that they are accepted by the BPF verifier may turn out to be an iterative and cumbersome task. Typically, a user will generate binary, check if verifier will accept it, and iterate over the process until the BPF verifier OKs it.

Using the proposed GAS infrastructure, the BPF target can issue helpful diagnostics to the user around some of the restrictions enforced by the verifier:

1. (Static size) No larger than BPF_MAXINSNS[12] insns.
2. There must be no unreachable code.
3. The assembler may also be able to alert the user if there exist some malformed jumps, specifically, jumps to an undefined target.

It is important to note that the above-mentioned list is a subset of restrictions imposed by the BPF verifier. Adding diagnostics for these in the BPF target will only reduce the pain, not eliminate it.

5 Next Steps and Future Work

Some of the identified tasks, not in order of priority or importance are:

1. Discuss and resolve the open questions around what should be done for those CFI directives cannot be synthesized by looking at the asm.
2. Discuss, design and implement SCFI for handling inline asm. The current command line option of `--scfi=all` will ignore all compiler generated CFI for the function containing the inlined assembly. Supporting a new argument like `--scfi=inline` requires that the compiler generated CFI be *not* dropped for functions with inline assembly.
3. Hardening of interfaces and design by adding one more architecture to the mix. `aarch64` seems to be good candidate. As noted earlier, the SCFI machinery has been implemented as a target-independent component in GAS. So ensuring that the SCFI machinery works for more than one ISA will be extremely helpful.
4. Testing with codebases with hand-written asm. Especially one with hand-written CFI annotations for its asm, so that there is something to cross-check the synthesized CFI against.

[11]Surely, not an exhaustive set; reviewer inputs for other useful diagnostics appreciated

[12]currently set to 4096

5. Add capability to synthesize CFI for functions using DRAP to realign stack.
6. Ensure decoupled `ginsn` creation from SCFI. As noted earlier, there are other use-cases for `ginsn` creation, like diagnostics for hand-written asm, specific validations, and even program verification for BPF.

Acknowledgments

The author would like to thank all the reviewers for their feedback.

References

[1] *The DWARF Debugging Information Format Standard*, Version 5, https://dwarfstd.org/dwarf5std.html

[2] *CFI directives in GNU AS*, https://sourceware.org/binutils/docs/as/CFI-directives.html

[3] *AArch64 Machine Directives in GNU AS* https://sourceware.org/binutils/docs/as/AArch64-Directives.html

[4] *binutils-gdb: gas: allow labeling of CFI instructions* https://sourceware.org/git/?p=binutils-gdb.git;a=commit;h=696025802ec3273fde5cbf82c215a3d795435c1a