Code Generation for Data Processing
Lecture 10: Unwinding and Debuginfo

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2023/24
Motivation: Meta-Information on Program

- Machine code suffices for execution → not true
- Needs program headers and entry point
- Linking with shared libraries needs dynamic symbols and interpreter
- Stack unwinding needs information about the stack
  - Size of each stack frame, destructors to be called, etc.
  - Vital for C++ exceptions, even for non-C++ code
- Stack traces require stack information to find return addresses
  - Use cases: coredumps, debuggers, profilers
- Debugging experience enhanced by variables, files, lines, statements, etc.
Adding Meta-Information with GCC

- **-g** supports different formats and levels (and GNU extensions)
- **-fexceptions**
- **-fasynchronous-unwind-tables**

- Exceptions must work without debuginfo
- Unwinding through code without exception-support must work
Stack Unwinding

- Needed for exceptions (_Unwind_RaiseException) or forced unwinding

- Search phase: walk through the stack, check whether to stop at each frame
  - May depend on exception type, ask *personality function*
  - Personality function needs extra language-specific data
  - Stop once an exception handler is found

- Cleanup phase: walk again, do cleanup and stop at handler
  - Personality function indicates whether handler needs to be called
  - Can be for exception handler or for calling destructors
  - If yes: personality function sets up registers/sp/po for landing pad
  - Non-matching handler or destructor-only: landing pad calls _Unwind_Resume
Stack Unwinding: Requirements

- Given: current register values in unwind function

- Need: iterate through stack frames
  - Get address of function of the stack frame
  - Get pc and sp for this function
  - Find personality function and language-specific data
  - Maybe get some registers from the stack frame
  - Update some registers with exception data

- Increased difficulty: stepping through signal handler
Stack Unwinding: setjmp/longjmp

- Simple idea – all functions that run code during unwinding do:
  - Register their handler at function entry
  - Deregister their handler at function exit
- Personality function sets jmpbuf to landing pad
- Unwinder does longjmp

+ Needs no extra information
- High overhead in non-exceptional case
Stack Unwinding: Frame Pointer

- Frame pointers allow for fast unwinding
- fp points to stored caller’s fp
- Return address stored adjacent to frame pointer

+ Fast and simple, also without exception
- Not all programs have frame pointers
  - Overhead of creating full stack frame
  - Causes loss of one register (esp. x86)
- Still needs to find meta-information
- Need to distinguish prologue with wrong info

x86_64:
```assembly
push rbp
mov rbp, rsp
// ...
mov rsp, rbp
pop rbp
ret
```
aarch64:
```assembly
stp x29, x30, [sp, -32]!
mov x29, sp
// ...
ldp x29, x30, [sp], 32
ret
```
Stack Unwinding: Without Frame Pointer

- Given: \( pc \) and \( sp \) (bottom of stack frame/call frame)
  - In parent frames: \( \text{retaddr} - 1 \sim pc \) and \( \text{CFA} \sim sp \)
- Need to map \( pc \) to stack frame size
  - \( sp + \text{framesize} = \text{CFA} \) (canonical frame address – \( sp \) at call)
  - Stack frame size varies throughout function, e.g. prologue

- Case 1: some register used as frame pointer – CFA constant offset to \( fp \)
  - E.g., for variable stack frame size
- Case 2: no frame pointer: CFA is constant offset to \( sp \)

\( \Rightarrow \) Unwinding \textit{must} restore register values
  - Other reg. can act as frame pointer, register saved in other register, \ldots
  - Need to know where return address is stored
Call Frame Information

- Table mapping each instr. to info about registers and CFA

- CFA: register with signed offset (or arbitrary expression)

- Register:
  - Undefined – unrecoverable (default for caller-saved reg)
  - Same – unmodified (default for callee-saved reg)
  - Offset(N) – stored at address CFA+N
  - Register(reg) – stored in other register
  - or arbitrary expressions
### Call Frame Information – Example 1

| foo:             | CFA   | rip     | rbx   | rbp   | ...
|------------------|-------|---------|-------|-------|-------
| 0x0: push rbx    |       | rsp+0x08|       | same  | same  |
| 0x1: mov ebx, edi|       | rsp+0x10|       | [CFA-0x10] | same  |
| 0x3: call bar    |       | rsp+0x10|       | [CFA-0x10] | same  |
| 0x8: mov eax, ebx|       | rsp+0x10| [CFA-0x08] | [CFA-0x10] | same  |
| 0xa: pop rbx     |       | rsp+0x10| [CFA-0x08] | [CFA-0x10] | same  |
| 0xb: ret         |       | rsp+0x08| [CFA-0x08] | same  | same  |
## Call Frame Information – Example 2

<table>
<thead>
<tr>
<th></th>
<th>CFA</th>
<th>rip</th>
<th>rbx</th>
<th>rbp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0:</td>
<td>push rbp</td>
<td>rsp+0x08</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x1:</td>
<td>mov rbp, rsp</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0x4:</td>
<td>shl rdi, 4</td>
<td>rbp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0x8:</td>
<td>sub rsp, rdi</td>
<td>rbp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0xb:</td>
<td>mov rdi, rsp</td>
<td>rbp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0xe:</td>
<td>call bar</td>
<td>rbp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0x13:</td>
<td>leave</td>
<td>rbp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>[CFA-0x10]</td>
</tr>
<tr>
<td>0x14:</td>
<td>ret</td>
<td>rsp+0x08</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
</tbody>
</table>
Call Frame Information – Example 3

<table>
<thead>
<tr>
<th>foo:</th>
<th>CFA</th>
<th>rip</th>
<th>rbx</th>
<th>rbp</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0:</td>
<td>sub rsp, 8</td>
<td>rsp+0x08</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x4:</td>
<td>test edi, edi</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x6:</td>
<td>js 0x12</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x8:</td>
<td>call positive</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0xd:</td>
<td>add rsp, 8</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x11:</td>
<td>ret</td>
<td>rsp+0x08</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x12:</td>
<td>call negative</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x17:</td>
<td>add rsp, 8</td>
<td>rsp+0x10</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
<tr>
<td>0x1a:</td>
<td>ret</td>
<td>rsp+0x08</td>
<td>[CFA-0x08]</td>
<td>same</td>
<td>same</td>
</tr>
</tbody>
</table>
Call Frame Information: Encoding

- Expanded table can be huge
- Contents change rather seldomly
  - Mainly in prologue/epilogue, but mostly constant in-between

- Idea: encode table as bytecode
- Bytecode has instructions to create a new row
  - Advance machine code location
- Bytecode has instructions to define CFA value
- Bytecode has instructions to define register location
- Bytecode has instructions to remember and restore state
## Call Frame Information: Bytecode – Example 1

<table>
<thead>
<tr>
<th></th>
<th>CFA</th>
<th>rip</th>
<th>rbx</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>foo:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>push rbx</td>
<td>rsp+8</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>1:</td>
<td>mov ebx, edi</td>
<td>rsp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>3:</td>
<td>call bar</td>
<td>rsp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>8:</td>
<td>mov eax, ebx</td>
<td>rsp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>a:</td>
<td>pop rbx</td>
<td>rsp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>b:</td>
<td>ret</td>
<td>rsp+8</td>
<td>[CFA-8]</td>
</tr>
</tbody>
</table>

- **DW_CFA_def_cfa:** RSP +8
- **DW_CFA_offset:** RIP -8
- **DW_CFA_advance_loc:** 1
- **DW_CFA_def_cfa_offset:** +16
- **DW_CFA_offset:** RBX -16
- **DW_CFA_advance_loc:** 10
- **DW_CFA_def_cfa_offset:** +8
### Call Frame Information: Bytecode – Example 2

<table>
<thead>
<tr>
<th></th>
<th>CFA</th>
<th>rip</th>
<th>rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>push rbp</td>
<td>rsp+8</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>1:</td>
<td>mov rbp, rsp</td>
<td>rsp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>4:</td>
<td>shl rdi, 4</td>
<td>rbp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>8:</td>
<td>sub rsp, rdi</td>
<td>rbp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>b:</td>
<td>mov rdi, rsp</td>
<td>rbp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>e:</td>
<td>call bar</td>
<td>rbp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>13:</td>
<td>leave</td>
<td>rbp+16</td>
<td>[CFA-8]</td>
</tr>
<tr>
<td>14:</td>
<td>ret</td>
<td>rsp+8</td>
<td>[CFA-8]</td>
</tr>
</tbody>
</table>

- **DW_CFA_def_cfa**: RSP +8
- **DW_CFA_offset**: RIP -8
- **DW_CFA_advance_loc**: 1
- **DW_CFA_def_cfa_offset**: +16
- **DW_CFA_offset**: RBP -16
- **DW_CFA_advance_loc**: 3
- **DW_CFA_def_cfa_register**: RBP
- **DW_CFA_advance_loc**: 16
- **DW_CFA_def_cfa**: RSP +8
### Call Frame Information: Bytecode – Example 3

<table>
<thead>
<tr>
<th></th>
<th>CFA</th>
<th>rip</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0:</td>
<td>sub rsp, 8</td>
<td>rsp+8  [CFA-8]</td>
</tr>
<tr>
<td>4:</td>
<td>test edi, edi</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>6:</td>
<td>js 0x12</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>8:</td>
<td>call positive</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>d:</td>
<td>add rsp, 8</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>11:</td>
<td>ret</td>
<td>rsp+8  [CFA-8]</td>
</tr>
<tr>
<td>12:</td>
<td>call negative</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>17:</td>
<td>add rsp, 8</td>
<td>rsp+16 [CFA-8]</td>
</tr>
<tr>
<td>1a:</td>
<td>ret</td>
<td>rsp+8  [CFA-8]</td>
</tr>
</tbody>
</table>

---

Remember stack: `{}`
Call Frame Information: Bytecode

- DWARF\(^{48}\) specifies bytecode for call frame information
- Self-contained section `.eh_frame` (or `.debug_frame`)
- Series of entries; two possible types distinguished using header

- Frame Description Entry (FDE): description of a function
  - Code range, instructions, pointer to CIE, language-specific data
- Common Information Entry (CIE): shared information among multiple FDEs
  - Initial instrs. (prepended to all FDE instrs.), personality function, alignment factors (constants factored out of instrs.), ...

- `readelf --debug-dump=frames <file>`
- `llvm-dwarfdump --debug-frame <file>`
Call Frame Information: .eh_frame_hdr

- Problem: linear search over – possibly many – FDEs is slow
- Idea: create binary search table over FDEs at link-time

- Ordered list of all function addresses and their FDE
- Unwinder does binary search to find matching FDE

- Separate program header entry: PT_GNU_EH_FRAME
- Unwinder needs loader support to find these
  - _dl_find_object or dl_iterate_phdr
- FDEs and indices are cached to avoid redundant lookups
Call Frame Information: Assembler Directives

- Compilers produces textual CFI
- Assembler encodes CFI into binary format
  - Allows for integration of annotated inline assembly
  - Inline-asm also needs CFI directives
- Register numbers specified by psABI
- Wrap function with `.cfi_startproc/.cfi_endproc`
- Many directives map straight to DWARF instructions
  - `.cfi_def_cfa_offset 16; .cfi_offset %rbp, -16;
    .cfi_def_cfa_register %rbp`
int bar(int*);
int foo(unsigned long x) {
    int arr[x * 4];
    return bar(arr);
}

gcc -O -S foo.c
Unwinding: Other Platforms

- Unwinding depends *strongly* on OS and architecture
- Linux uses DWARF
- Apple has modified version
- Windows has SEH with kernel-support for unwinding
- IBM AIX has their own format
- AArch32 has another custom format

- Additionally: minor differences for return address, stack handling, ...
Debugging: Wanted Features

- Get back trace
- Map address to source file/line
- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum
- Set break point at line/function
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement
Line Table

- Map instruction to: file/line/column; start of stmt; start of basic block; is prologue/epilogue; ISA mode
- Table can be huge; idea: encode as bytecode
- Extracted information are bytecode registers
- Conceptually similar to CFI encoding

- `llvm-dwarfdump -v --debug-line` or `readelf -wlL`
Debugging: Wanted Features

- Get back trace
- Map address to source file/line
- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum
- Set break point at line/function
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement
DWARF: Hierarchical Program Description

- Extensible, flexible, Turing-complete\(^49\) format to describe program

- Forest of Debugging Information Entries (DIEs)
  - Tag: indicates what the DIE describes
  - Set of attributes: describe DIE (often constant, range, or arbitrary expression)
  - Optionally children

- Rough classification:
  - DIEs for types: base types, typedef, struct, array, enum, union, ...
  - DIEs for data objects: variable, parameter, constant
  - DIEs for program scope: compilation unit, function, block, ...

DWARF: Data Types

```
DW_TAG_structure_type [0x2e]
    DW_AT_byte_size (0x08)
    DW_AT_sibling (0x4a)
    DW_TAG_member [0x37]
        DW_AT_name ("x")
        DW_AT_type (0x4a "int")
        DW_AT_data_member_location (0x00)
    DW_TAG_member [0x40]
        DW_AT_name ("y")
        DW_AT_type (0x4a "int")
        DW_AT_data_member_location (0x04)

DW_TAG_base_type [0x4a]
    DW_AT_byte_size (0x04)
    DW_AT_encoding (DW_ATE_signed)
    DW_AT_name ("int")

```

```
DW_TAG_pointer_type [0xb1]
    DW_AT_byte_size (8)
    DW_AT_type (0xb6 "char *")

```

```
DW_TAG_pointer_type [0xb6]
    DW_AT_byte_size (8)
    DW_AT_type (0xbb "char")

```

```
DW_TAG_base_type [0xbb]
    DW_AT_byte_size (0x01)
    DW_AT_encoding (DW_ATE_signed_char)
    DW_AT_name ("char")
```
DWARF: Variables

DW_TAG_variable [0xa3]
  DW_AT_name          ("x")
  DW_AT_decl_file    ("/path/to/main.c")
  DW_AT_decl_line    (2)
  DW_AT_decl_column  (0x2e)
  DW_AT_type         (0x4a "int")
  DW_AT_location     (0x3b:
                       [0x08, 0x0c): DW_OP_breg3 RBX+0, DW_OP_lit1, DW_OP_shl, DW_OP_stack_value
                       [0x0c, 0x0d): DW_OP_entry_value(DW_OP_reg5 RDI), DW_OP_lit1, \
                               DW_OP_shl, DW_OP_stack_value)

DW_TAG_formal_parameter [0x7f]
  DW_AT_name          ("argc")
  // ...
DWARF: Expressions

- Very general way to describe location of value: bytecode

- Stack machine, evaluates to location or value of variable
  - Simple case: register or stack slot
  - But: complex expression to recover original value after optimization
    e.g., able to recover $i$ from stored $i - 1$
  - Unbounded complexity!

- Can contain control flow
- Can dereference memory, registers, etc.

- Used for: CFI locations, variable locations, array sizes, ...
DWARF: Program Structure

- Follows structure of code
- Top-level: compilation unit
- Entries for namespaces, subroutines (functions)
  - Functions can contain inlined subroutines
- Lexical blocks to group variables
- Call sites and parameters
- Each node annotated with pc-range and source location
Debugging: Wanted Features

- Get back trace
- Map address to source file/line
- Show global and local variables
  - Local variables need scope information, e.g. shadowing
  - Data type information, e.g. int, string, struct, enum
- Set break point at line/function
  - Might require multiple actual breakpoints: inlining, template expansion
- Step through program by line/statement
Other Debuginfo Formats

- DWARF is big despite compression
- Cannot run in time-constrained environments
  - Uns suited for in-kernel backtrace generation
- Historically: STABS – string based encoding
  - Complexity increased significantly over time
- Microsoft: PDB for PE
- Linux kernel: CTF for simple type information
- Linux kernel: BTF for BPF programs
Some languages/setups must be able to unwind the stack

- Needs meta-information on call frames
- DWARF encodes call frame information is bytecode program
- Runtime must efficiently find relevant information
- Stack unwinding typically done in two phases
- Functions have associated personality function to steer unwinding
- DWARF encodes debug info in tree structure of DIEs
- DWARF info can become arbitrarily complex
Unwinding and Debuginfo – Questions

- What are alternatives to stack unwinding?
- What are the benefits of stack unwinding through metadata?
- What are the two phases of unwinding? Why is this separated?
- How to construct a CFI table for a given assembly code?
- How to construct DWARF ops for a CFI table?
- How to find the correct CFI table line for a given address?
- What is the general structure of DWARF debug info?