

# 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 2024/25

# 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`  
`-fexceptions`  
`-fasynchronous-unwind-tables`

- ▶ `-g` supports different formats and levels (and GNU extensions)
- ▶ 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/pc 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

## 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)
- Not generally possible to restore callee-saved registers
- ▶ Still needs to find meta-information

```
x86_64:  
    push rbp  
    mov rbp, rsp  
    // ...  
    mov rsp, rbp  
    pop rbp  
    ret
```

```
aarch64:  
    stp x29, x30, [sp, -32]!  
    mov x29, sp  
    // ...  
    ldp x29, x30, [sp], 32  
    ret
```

## Stack Unwinding: Without Frame Pointer

- ▶ Definition: *canonical frame address (CFA)* is  $sp$  at the function call
  - ▶ Given:  $pc$  and  $sp$  (bottom of stack frame/call frame)
    - ▶ In parent frames:  $retaddr - 1 \sim pc$  and  $CFA \sim sp$
  - ▶ Need to map  $pc$  to stack frame size
    - ▶  $sp + framesize = CFA$
    - ▶ Stack frame size varies throughout function, e.g. prologue, stack arguments
  - ▶ Case 1: some register used as frame pointer – CFA constant offset to  $fp$ 
    - ▶ E.g., for variable stack frame size, stack realignment on function entry
  - ▶ Case 2: no frame pointer: CFA is constant offset to  $sp$
- ↪ Unwinding *must* restore register values
- ▶ Other reg. can act as frame pointer, register saved in other register, ...
  - ▶ 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

	CFA	rip	rbx	rbp	...
foo:					
0x0: push rbx	rsp+0x08	[CFA-0x08]	same	same	
0x1: mov ebx, edi	rsp+0x10	[CFA-0x08]	[CFA-0x10]	same	
0x3: call bar	rsp+0x10	[CFA-0x08]	[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

		CFA	rip	rbx	rbp	...
	foo:					
0x0:	push rbp	rsp+0x08	[CFA-0x08]	same	same	
0x1:	mov rbp, rsp	rsp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0x4:	shl rdi, 4	rbp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0x8:	sub rsp, rdi	rbp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0xb:	mov rdi, rsp	rbp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0xe:	call bar	rbp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0x13:	leave	rbp+0x10	[CFA-0x08]	same	[CFA-0x10]	
0x14:	ret	rsp+0x08	[CFA-0x08]	same	same	

## Call Frame Information – Example 3

		CFA	rip	rbx	rbp	...
	foo:					
0x0:	sub rsp, 8	rsp+0x08	[CFA-0x08]	same	same	
0x4:	test edi, edi	rsp+0x10	[CFA-0x08]	same	same	
0x6:	js 0x12	rsp+0x10	[CFA-0x08]	same	same	
0x8:	call positive	rsp+0x10	[CFA-0x08]	same	same	
0xd:	add rsp, 8	rsp+0x10	[CFA-0x08]	same	same	
0x11:	ret	rsp+0x08	[CFA-0x08]	same	same	
0x12:	call negative	rsp+0x10	[CFA-0x08]	same	same	
0x17:	add rsp, 8	rsp+0x10	[CFA-0x08]	same	same	
0x1a:	ret	rsp+0x08	[CFA-0x08]	same	same	

## Call Frame Information – Exercise

- ▶ Download `ex10.txt` from the course website
- ▶ Construct the CFI tables for both functions  
(you can omit lines that don't change)

## 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

	CFA	rip	rbx	
				DW_CFA_def_cfa: RSP +8
				DW_CFA_offset: RIP -8
				DW_CFA_advance_loc: 1
0: push rbx	rsp+8	[CFA-8]		DW_CFA_def_cfa_offset: +16
1: mov ebx, edi	rsp+16	[CFA-8]	[CFA-16]	DW_CFA_offset: RBX -16
3: call bar	rsp+16	[CFA-8]	[CFA-16]	DW_CFA_advance_loc: 10
8: mov eax, ebx	rsp+16	[CFA-8]	[CFA-16]	DW_CFA_def_cfa_offset: +8
a: pop rbx	rsp+16	[CFA-8]	[CFA-16]	
b: ret	rsp+8	[CFA-8]	[CFA-16]	

## Call Frame Information: Bytecode – Example 2

	CFA	rip	rbp	
				DW_CFA_def_cfa: RSP +8
				DW_CFA_offset: RIP -8
				DW_CFA_advance_loc: 1
foo:				DW_CFA_def_cfa_offset: +16
0: push rbp	rsp+8	[CFA-8]		DW_CFA_offset: RBP -16
1: mov rbp, rsp	rsp+16	[CFA-8]	[CFA-16]	DW_CFA_advance_loc: 3
4: shl rdi, 4	rbp+16	[CFA-8]	[CFA-16]	DW_CFA_def_cfa_register: RBP
8: sub rsp, rdi	rbp+16	[CFA-8]	[CFA-16]	DW_CFA_advance_loc: 16
b: mov rdi, rsp	rbp+16	[CFA-8]	[CFA-16]	DW_CFA_def_cfa: RSP +8
e: call bar	rbp+16	[CFA-8]	[CFA-16]	
13: leave	rbp+16	[CFA-8]	[CFA-16]	
14: ret	rsp+8	[CFA-8]	[CFA-16]	



## Call Frame Information: Bytecode – Example 3

	CFA	rip	
foo:			DW_CFA_def_cfa: RSP +8
0: sub rsp, 8	rsp+8	[CFA-8]	DW_CFA_offset: RIP -8
4: test edi, edi	rsp+16	[CFA-8]	DW_CFA_advance_loc: 4
6: js 0x12	rsp+16	[CFA-8]	DW_CFA_def_cfa_offset: +16
8: call positive	rsp+16	[CFA-8]	DW_CFA_advance_loc: 13
d: add rsp, 8	rsp+16	[CFA-8]	DW_CFA_remember_state:
11: ret	rsp+8	[CFA-8]	DW_CFA_def_cfa_offset: +8
12: call negative	rsp+16	[CFA-8]	DW_CFA_advance_loc: 1
17: add rsp, 8	rsp+16	[CFA-8]	DW_CFA_restore_state:
1a: ret	rsp+8	[CFA-8]	DW_CFA_advance_loc: 9
			DW_CFA_def_cfa_offset: +8

Remember stack: {}

## Call Frame Information: Bytecode – Exercise

- ▶ For the functions in `ex10.txt`:  
encode your CFI tables in DWARF CFI bytecode
- ▶ Can you reduce the size of the bytecode by changing or omitting instructions while maintaining correctness?

## Call Frame Information: Bytecode

- ▶ DWARF<sup>52</sup> 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`<sup>53</sup>

- ▶ 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

<sup>53</sup>[https://refspecs.linuxfoundation.org/LSB\\_4.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html](https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html)

## 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`

# Call Frame Information: Assembler Directives – Example

```
int bar(int*);  
int foo(unsigned long x) {  
    int arr[x * 4];  
    return bar(arr);  
}
```

```
gcc -O -S foo.c
```

```
                .globl foo  
                .type foo, @function  
  
foo:  
  
                .cfi_startproc  
                push rbp  
                .cfi_def_cfa_offset 16  
                .cfi_offset 6, -16  
                mov rbp, rsp  
                .cfi_def_cfa_register 6  
                shl rdi, 4  
                sub rsp, rdi  
                mov rdi, rsp  
                call bar  
                leave  
                .cfi_def_cfa 7, 8  
                ret  
                .cfi_endproc  
                .size foo, .-foo
```

# Unwinding from Signal Handler

- ▶ Unwinding is conceptually supported even from signal handlers
- ↪ Possible to get backtraces in-program in signal handler
- ▶ Unwind info must be correct at every single instruction (“asynchronous”)
  - ▶ Otherwise, it only needs to be correct at calls (“synchronous unwinding”)
- ▶ Is throwing exceptions from signal handlers safe? No!
  - ▶ Variables can be in an inconsistent state, e.g. in the middle of a copy
  - ▶ Possible and viable only under very limited and controlled circumstances

## Unwinding: Other Platforms

- ▶ Unwinding depends *strongly* on OS and architecture
- ▶ GNU/Linux uses DWARF
- ▶ Apple has modified compact 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, ...

---

Needs to work reliably for exception handling



# Debugging: Wanted Features

- ▶ Get back trace ↔ CFI
- ▶ 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

# Debug Frame Information

- ▶ `.debug_frame` is very similar to `.eh_frame`
- ▶ Caveat: there are subtle encoding differences
- ▶ `eh_frame` allows for some (GNU) extensions

# Line Table


- ▶ Map instruction to: file/line/column and ISA mode
- ▶ Also: mark start of stmt; start of basic block; prologue end/epilogue begin
  - ▶ Provide breakpoint hints for lines, function entry/exit
- ▶ 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 ↔ CFI
- ▶ Map address to source file/line ↔ Line Table
- ▶ 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 ↔ Line Table/??
  - ▶ Might require multiple actual breakpoints: inlining, template expansion
- ▶ Step through program by line/statement ↔ Line Table

# DWARF: Hierarchical Program Description

- ▶ Extensible, flexible, Turing-complete<sup>54</sup> 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, ...

<sup>54</sup>J Oakley and S Bratus. "Exploiting the Hard-Working DWARF: Trojan and Exploit Techniques with No Native Executable Code". In: *WOOT*. 2011. 

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

## Other Debuginfo Formats

- ▶ DWARF is big despite compression
- ▶ Cannot run in time-constrained environments
  - ▶ Unsited 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

# Unwinding and Debuginfo – Summary

- ▶ Some languages/setups must be able to unwind the stack
- ▶ Needs meta-information on call frames
- ▶ DWARF encodes call frame information in 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?