Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

Towards Data Type Profiling

Namhyung Kim <namhyung@kernel.org>
Google

Data type profiling

- Goal
  - Precise memory access profile with type info
  - Help memory layout optimization
  - No changes in the target code

- How?
  - PMU[1] precise sampling (by Linux perf tools)
  - DWARF[2] location description

1. PMU: Performance Monitoring Unit in CPU cores (or other units)
2. DWARF: Debugging With Arbitrary Record Formats. The standard debug format in Linux

Existing memory profilers

- PMU sample based
  - perf mem - utilize data source (and more) in the PMU sample
  - perf c2c - dedicate to check data (false) sharing

- Heap allocation based
  - Heaptrack - leak check, heap usage, temporary allocation, ...
  - Valgrind - leak/undefined access check, cache simulation, heap usage, ...

PMU precise memory sampling

- Modern Processors provide precise memory access information like
  - Instruction address
  - Data address
  - Data source (L1$, L2$, L3$, memory, ...)
  - Latency
  - ...

- Supported vendors: Intel (PEBS[1]), AMD (IBS[2]), ARM (SPE[3]), ...

1. Processor Event-Based Sampling
2. Instruction Based Sampling
3. Statistical Profiling Extension

PMU precise memory events

Each vendor has different capabilities:

- Intel (PEBS)
  - Sample memory operations (load or store) only
  - Loads can have a latency filter (threshold)
- AMD (IBS)
  - Sample any operations (uops) without filtering
  - Only memory u-ops will have meaningful info
- ARM (SPE)
  - Sample any operations with filtering
  - Can filter load and/or store operations, with latency filter too

Recording PMU precise memory samples

- Simply use **perf mem record**

- For advanced users

```
perf mem record -t load       # load operation only (Intel, ARM)
perf mem record --ldlat=10    # load latency filter (Intel, ARM)
perf mem record -K            # for kernel only (Intel, ARM)
perf record -e $EVENT         # if you know what you do
```

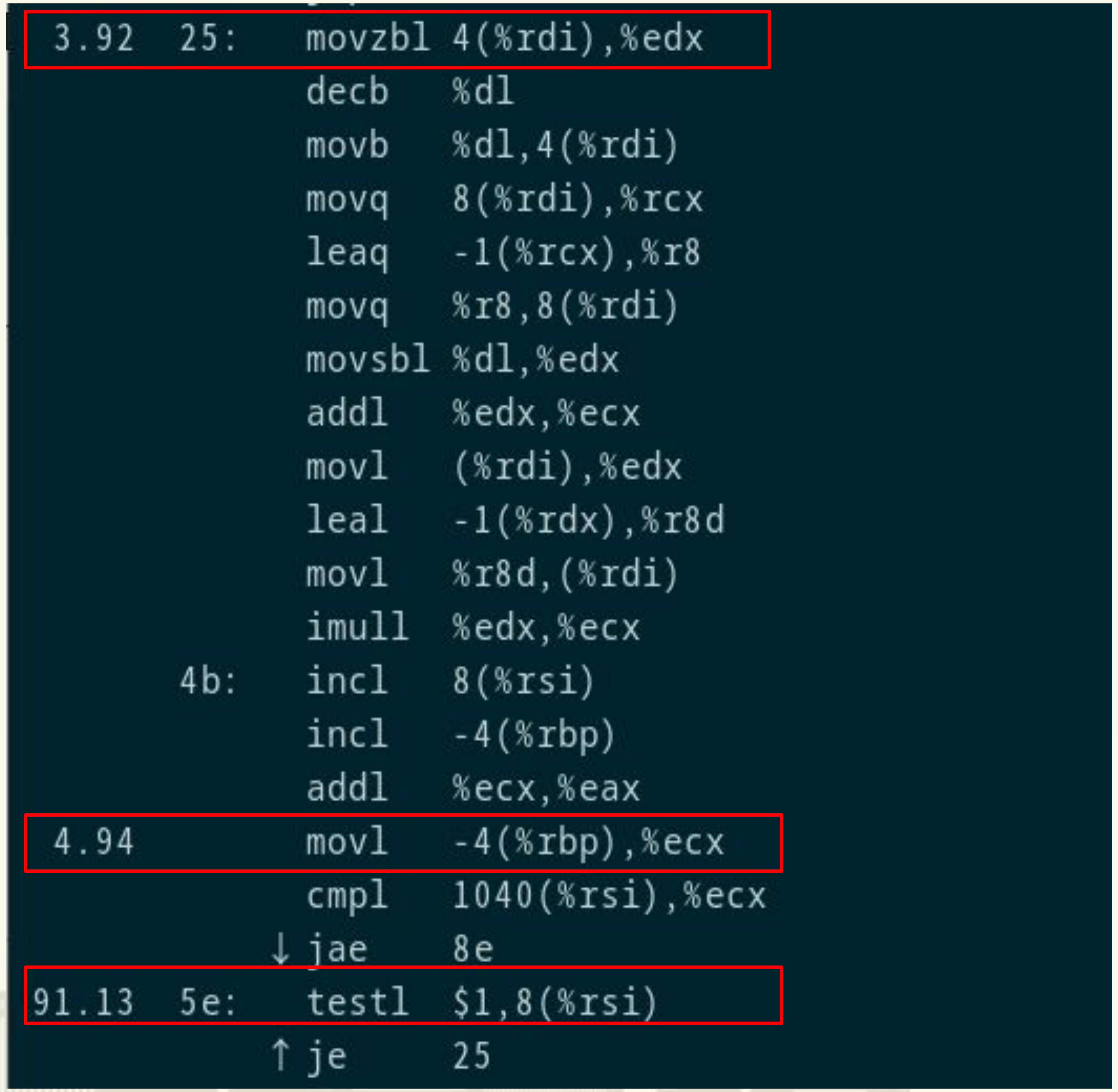
Getting memory location

perf mem record $PROG

perf annotate

- register
- offset

overhead(%) offset: instructions



```
 3.92   25:    movzbl 4(%rdi),%edx
               decb   %dl
               movb   %dl,4(%rdi)
               movq   8(%rdi),%rcx
               leaq   -1(%rcx),%r8
               movq   %r8,8(%rdi)
               movsbl %dl,%edx
               addl   %edx,%ecx
               movl   (%rdi),%edx
               leal   -1(%rdx),%r8d
               movl   %r8d,(%rdi)
               imull  %edx,%ecx
        4b:    incl   8(%rsi)
               incl   -4(%rbp)
               addl   %ecx,%eax
 4.94          movl   -4(%rbp),%ecx
               cmpl   1040(%rsi),%ecx
             ↓ jae    8e
91.13   5e:    testl  $1,8(%rsi)
             ↑ je     25
```

4(%rdi)

-4(%rbp)

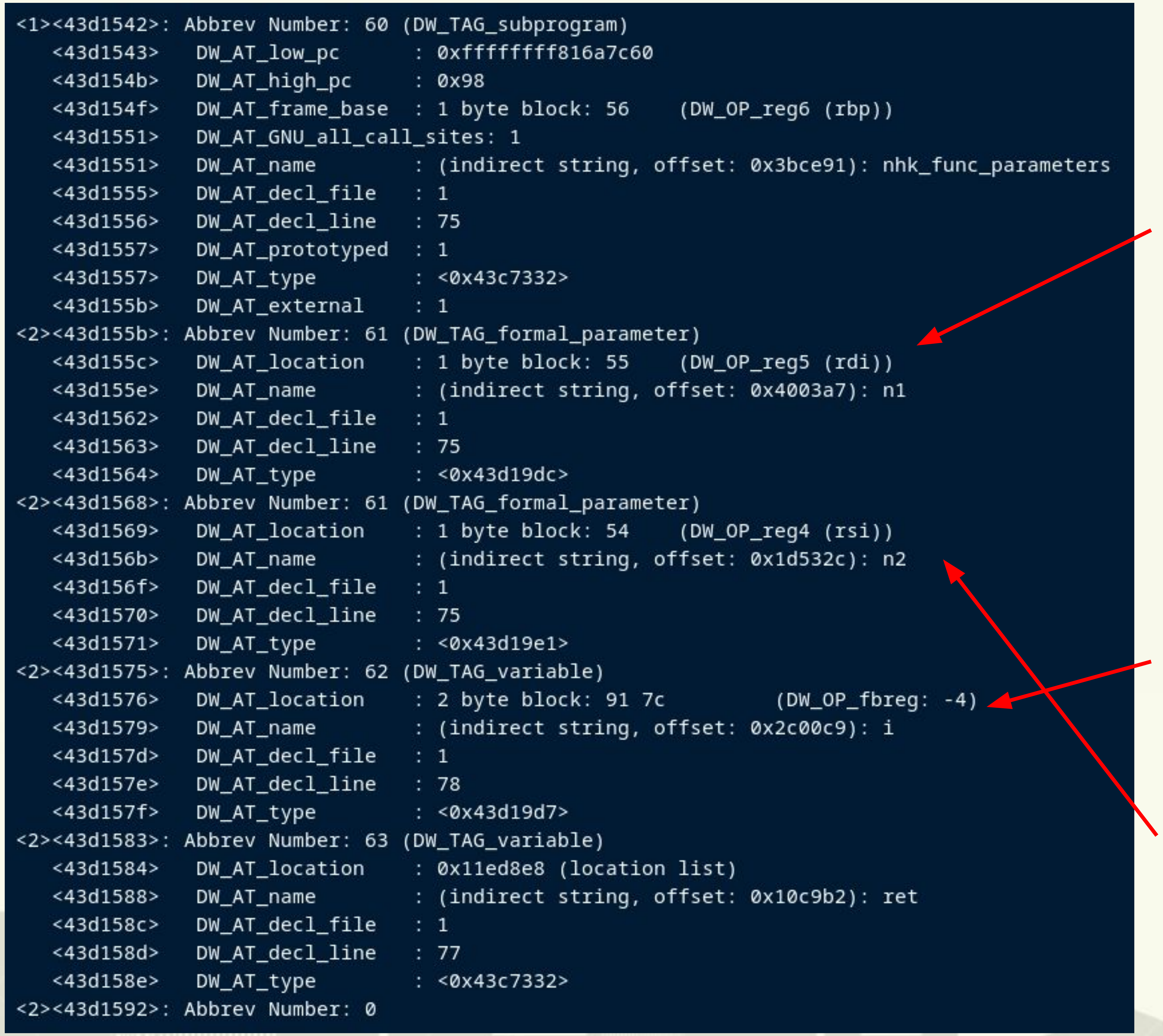
8(%rsi)

Location expression

readelf -wi

Debug info in

- variable
- parameter



```
<1><43d1542>: Abbrev Number: 60 (DW_TAG_subprogram)
   <43d1543>   DW_AT_low_pc      : 0xffffffff816a7c60
   <43d154b>   DW_AT_high_pc     : 0x98
   <43d154f>   DW_AT_frame_base  : 1 byte block: 56     (DW_OP_reg6 (rbp))
   <43d1551>   DW_AT_GNU_all_call_sites: 1
   <43d1551>   DW_AT_name        : (indirect string, offset: 0x3bce91): nhk_func_parameters
   <43d1555>   DW_AT_decl_file   : 1
   <43d1556>   DW_AT_decl_line   : 75
   <43d1557>   DW_AT_prototyped  : 1
   <43d1557>   DW_AT_type        : <0x43c7332>
   <43d155b>   DW_AT_external    : 1
<2><43d155b>: Abbrev Number: 61 (DW_TAG_formal_parameter)
   <43d155c>   DW_AT_location    : 1 byte block: 55     (DW_OP_reg5 (rdi))
   <43d155e>   DW_AT_name        : (indirect string, offset: 0x4003a7): n1
   <43d1562>   DW_AT_decl_file   : 1
   <43d1563>   DW_AT_decl_line   : 75
   <43d1564>   DW_AT_type        : <0x43d19dc>
<2><43d1568>: Abbrev Number: 61 (DW_TAG_formal_parameter)
   <43d1569>   DW_AT_location    : 1 byte block: 54     (DW_OP_reg4 (rsi))
   <43d156b>   DW_AT_name        : (indirect string, offset: 0x1d532c): n2
   <43d156f>   DW_AT_decl_file   : 1
   <43d1570>   DW_AT_decl_line   : 75
   <43d1571>   DW_AT_type        : <0x43d19e1>
<2><43d1575>: Abbrev Number: 62 (DW_TAG_variable)
   <43d1576>   DW_AT_location    : 2 byte block: 91 7c          (DW_OP_fbreg: -4)
   <43d1579>   DW_AT_name        : (indirect string, offset: 0x2c00c9): i
   <43d157d>   DW_AT_decl_file   : 1
   <43d157e>   DW_AT_decl_line   : 78
   <43d157f>   DW_AT_type        : <0x43d19d7>
<2><43d1583>: Abbrev Number: 63 (DW_TAG_variable)
   <43d1584>   DW_AT_location    : 0x11ed8e8 (location list)
   <43d1588>   DW_AT_name        : (indirect string, offset: 0x10c9b2): ret
   <43d158c>   DW_AT_decl_file   : 1
   <43d158d>   DW_AT_decl_line   : 77
   <43d158e>   DW_AT_type        : <0x43c7332>
<2><43d1592>: Abbrev Number: 0
```

4(%rdi)

-4(%rbp)

8(%rsi)

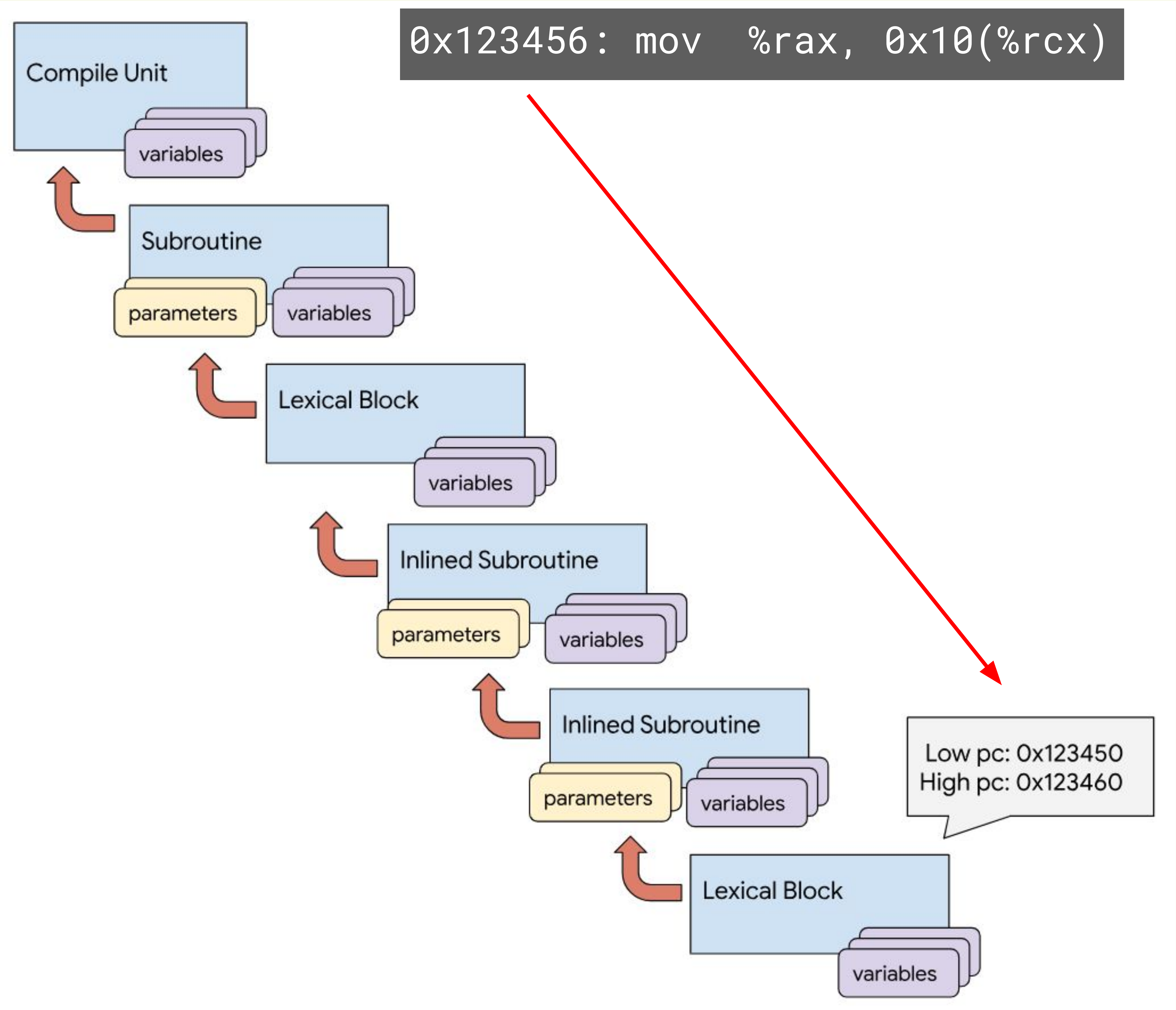
DWARF location description

- Location expression
  - Stack machine to specify a location
  - Register / mem / arithmetic operations / stack operations / ...

- Location list
  - When a variable is moving around different places (e.g. stack spill)
  - List of (code range + location expression)

Getting DWARF info

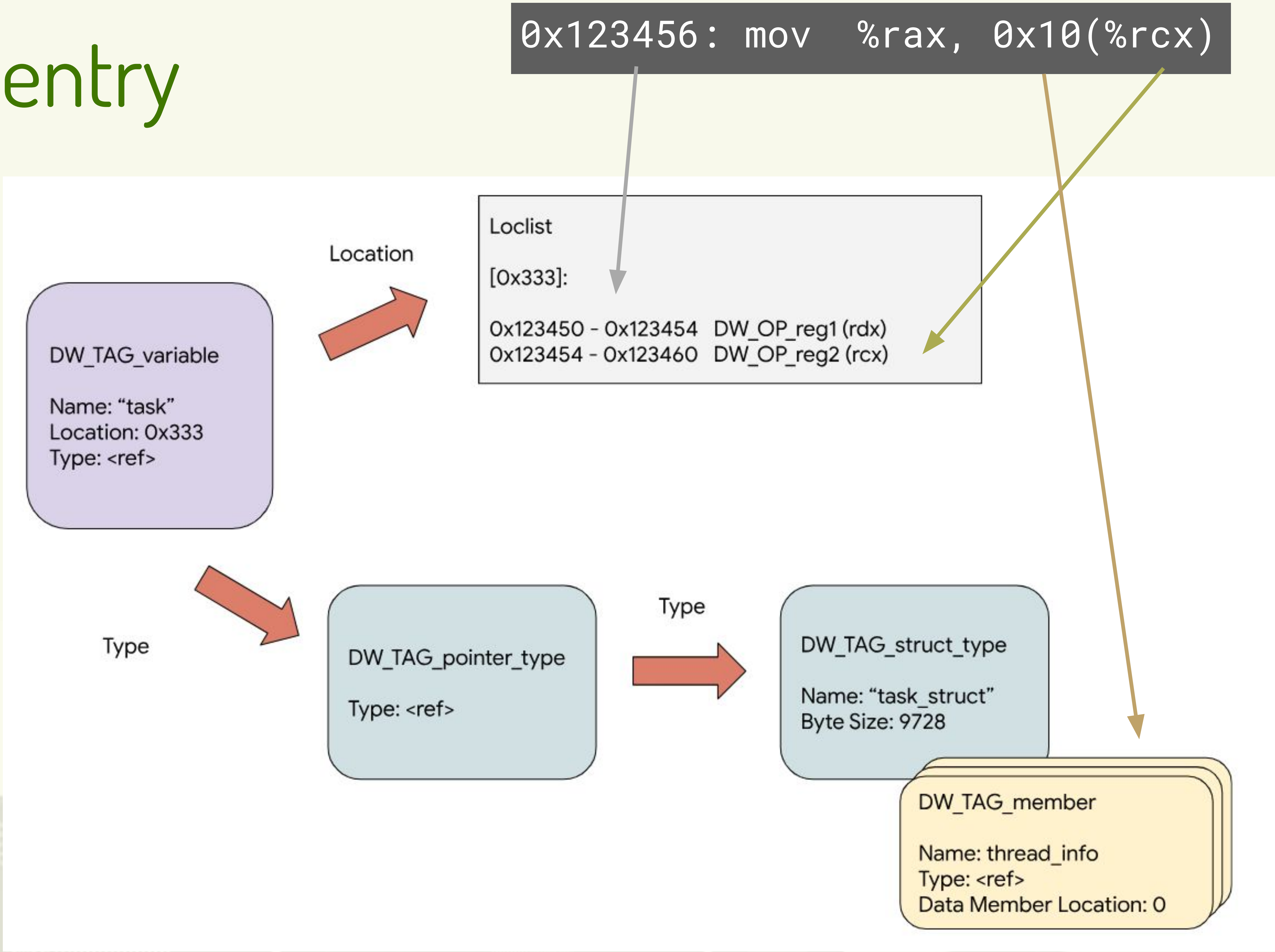
- Tree-like structure:
  - Find nested scopes using instruction address
- Each scope entry would have
  - Low PC and High PC attributes for the containing address range
  - Or, range list for scattered ranges



```
0x123456: mov  %rax, 0x10(%rcx)
```

Debug info of a variable entry

- Location list has
  - Code range
  - Location (reg/mem)
- Type info has
  - Name, kind
  - Member type / offset

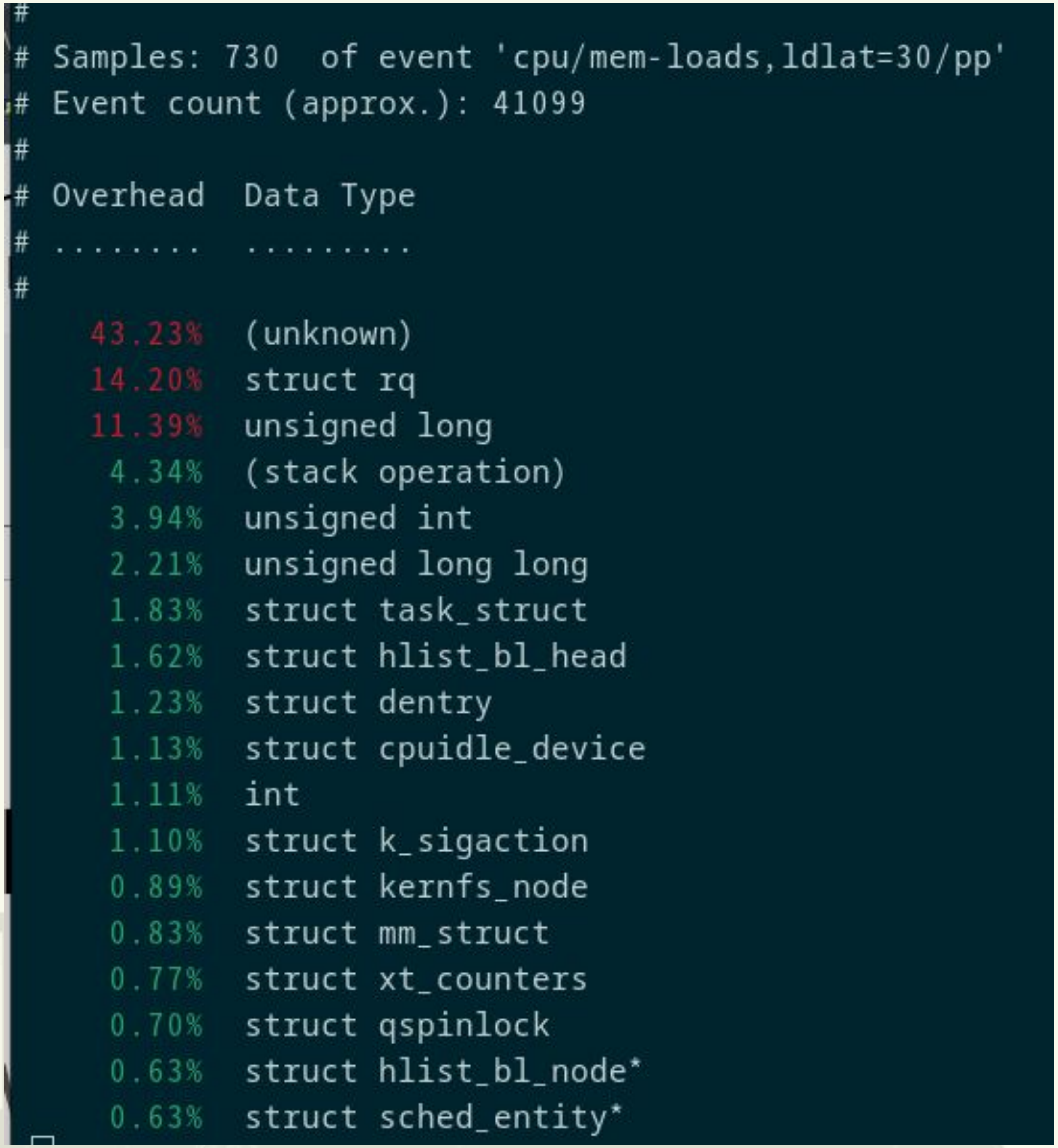


```
0x123456: mov  %rax, 0x10(%rcx)
```

Result: perf report -s type

What it does:

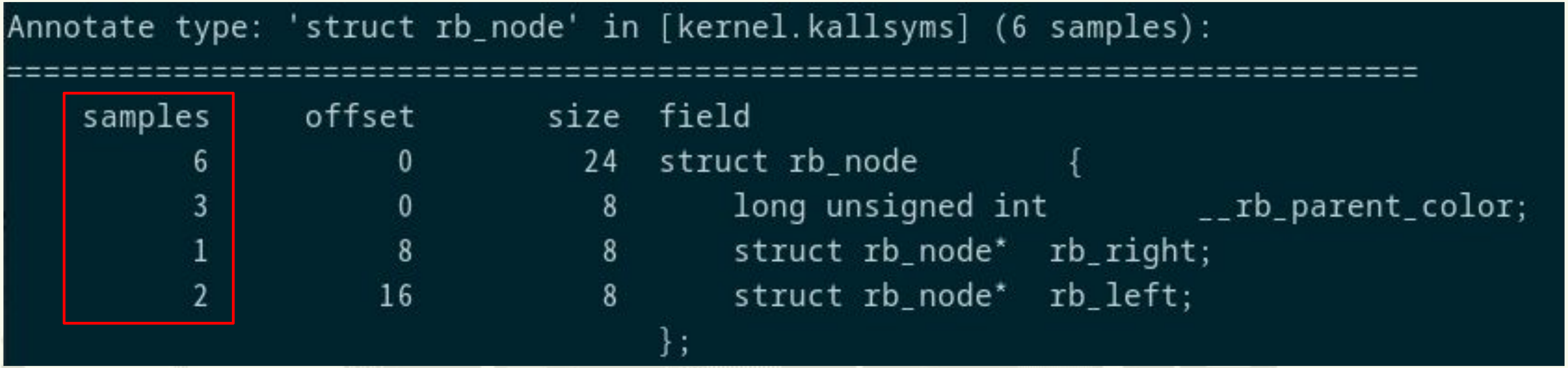
1. Identify an instruction from a sample
2. Extract a register from the instruction
3. Find a variable for matching register
4. Get the type of the variable
5. Aggregate the result for the type



```
#
# Samples: 730  of event 'cpu/mem-loads,ldlat=30/pp'
# Event count (approx.): 41099
#
# Overhead  Data Type
# ........  .........
#
    43.23%  (unknown)
    14.20%  struct rq
    11.39%  unsigned long
     4.34%  (stack operation)
     3.94%  unsigned int
     2.21%  unsigned long long
     1.83%  struct task_struct
     1.62%  struct hlist_bl_head
     1.23%  struct dentry
     1.13%  struct cpuidle_device
     1.11%  int
     1.10%  struct k_sigaction
     0.89%  struct kernfs_node
     0.83%  struct mm_struct
     0.77%  struct xt_counters
     0.70%  struct qspinlock
     0.63%  struct hlist_bl_node*
     0.63%  struct sched_entity*
```

Result: perf annotate --data-type

- Same approach + Use offset info to identify the field
  - **perf report** also has ‘typeoff’ sort key to show per-field overheads



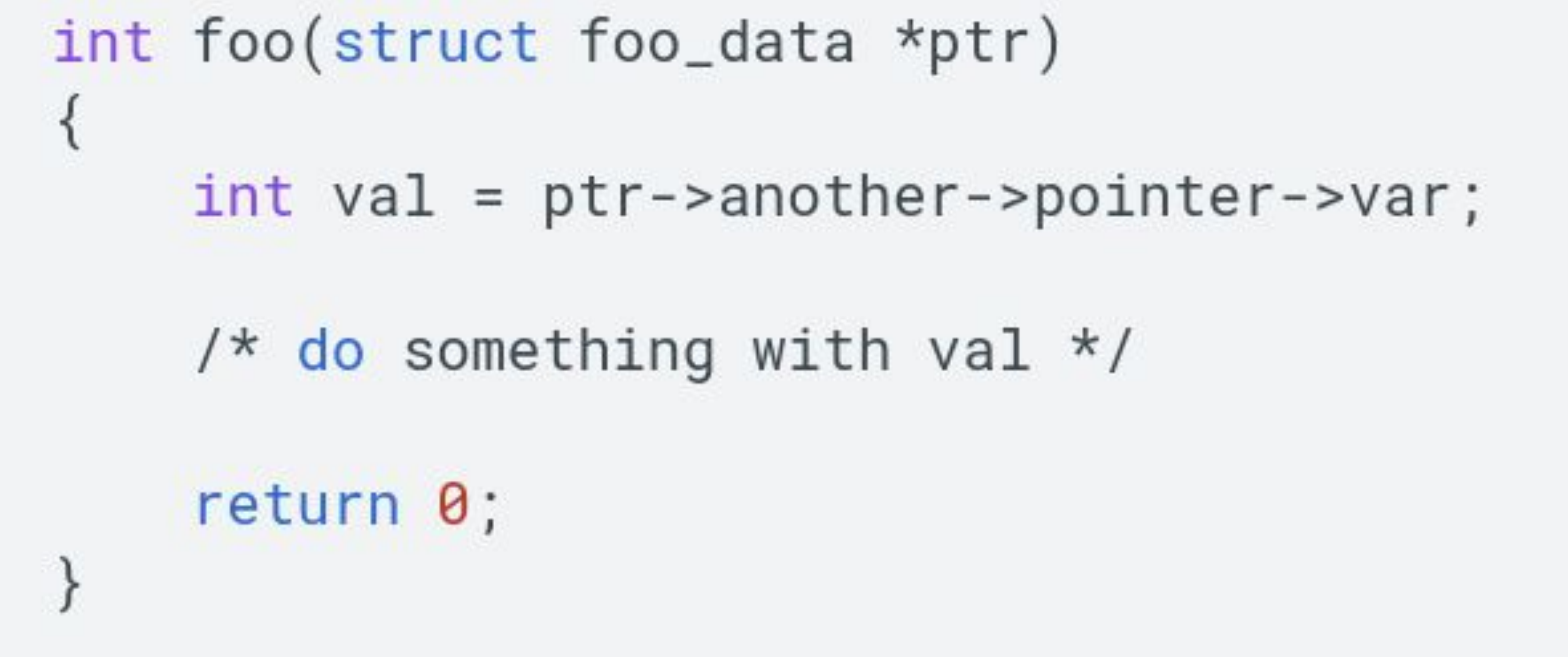
```
Annotate type: 'struct rb_node' in [kernel.kallsyms] (6 samples):
=====================================================================
   samples     offset      size  field
         6          0        24  struct rb_node       {
         3          0         8      long unsigned int        __rb_parent_color;
         1          8         8      struct rb_node*  rb_right;
         2         16         8      struct rb_node*  rb_left;
                                 };
```

Issues

- No variables
- Compiler optimizations
- Struct layout randomization
- Per-cpu (kernel) or TLS[1] (user) access
- Split DWARF support
- Languages
- Performance
- And more... ?

[1] TLS: Thread Local Storage

No variables: chain of pointers



```c
int foo(struct foo_data *ptr)
{
    int val = ptr->another->pointer->var;

    /* do something with val */

    return 0;
}
```

```
<foo>:
0x000100:   push  %rbp
0x000101:   mov   %rsp, %rbp
0x000104:   mov   0x0(%rdi), %rdx     ; ptr->another
0x000108:   mov   0x8(%rdx), %rcx     ; another->pointer
0x00010c:   movl  0x10(%rcx), %eax    ; pointer->var
…
```

DWARF (.debug_info)

```
DW_TAG_subprogram (foo)
  DW_TAG_formal_parameter (ptr)
    DW_AT_location (%rdi)
  DW_TAG_variable (val)
    DW_AT_location (fbreg -4)
```

What's in %rcx?

No variables: Possible solutions #1

- Build a full location table (in perf tool)
  - Go through the instructions and propagate the variable types
  - Follow pointer dereferences (a->b->c ...)

	reg1	reg2	reg3	...
pc1	type1	N/A	type2	
pc2	same	type3	same	
...	...	...	N/A	

No variables: Possible solutions #2

- Compiler can generate more information
  - Insert an artificial debug entry (short term)
    - For chains of pointers (and type casts too?)
    - With proper location expression and type info
  - Inverted location list (long term)
    - suggested in the DWARF discuss list
    - https://lists.dwarfstd.org/pipermail/dwarf-discuss/2023-June/002278.html

Compiler optimizations

- Compilers can change struct layouts
  - SROA[1] for local variables (pointer not taken?)
  - Currently perf rejects complex location expressions

- What can it accept?
  - a pointer variable is in a register
  - static memory location for global variables
  - stack location from the frame base for local variables

[1] SROA: Scalar Replacement of Aggregates

Struct layout randomization

- Sounds scary!
  - compiler plugin to randomize some structures
    - CONFIG_RANDSTRUCT
    - basically for structs with function pointers only?
  - hope it'd update DWARF location expression
    - Haven't tested it yet

Language support

- The first target is C
  - Kernel on x86
  - C issues: union, array, bitfield, type cast, return value, ...

- For userspace support
  - Support for other languages: C++, Rust, Go, ...
  - Never tried yet

Per-cpu variables in kernel

- Per-cpu variable in the kernel
  - Each cpu has its own copy of the variable
  - TLS[1] for user binaries would have similar concerns

- Variables can have complex(?) location expressions
  - __per_cpu_offset[cpu] + variable address
  - %gs: variable address (for this cpu)

[1] TLS: Thread Local Storage

Split DWARF

- DWARF4 + fission or DWARF5
  - How well is it supported?
  - perf uses elfutils/libdw

Performance issues

- Objdump on kernel
  - To get assembly code
  - GNU objdump with debug info is very slower than LLVM
  - LLVM objdump without debug info is slightly slower then GNU

- Use in-kernel instruction decoder (x86)
  - To extract location info from the instruction

Summary

- Perf tools implement data type profiling using PMU and DWARF
- Need more toolchain supports to produce better DWARF
- Let's make it more useful and easy to use!

- Links
  - v1: https://lore.kernel.org/lkml/20231012035111.676789-1-namhyung@kernel.org/
  - v2: https://lore.kernel.org/lkml/20231110000012.3538610-1-namhyung@kernel.org/

Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023

LPC 2023 - Overview

Conference Details

The Linux Plumbers Conference is the premier event for developers working at all levels of the plumbing layer and beyond.

Taking place on Monday 13th, Tuesday 14th and Wednesday 15th of November, this year we will be both in person and remote (hybrid). However to minimize technical issues, we'd appreciate most of the content presenters being in-person.

The in-person venue is the Omni Richmond Hotel, Richmond, VA.

100 S 12th St, Richmond, VA 23219, United States

Unless specified otherwise, the conference information will be shared in Eastern Standard Time (EST, UTC-05:00, America/EST timezone).

Sponsorship opportunities

Linux Plumbers Conference would not be possible without our sponsors. Many thanks to all the great organizations that have supported Linux Plumbers Conference over the years.

New sponsorship opportunities are available for 2023! We hope that your organization will consider joining our growing list of amazing sponsors this year. Find out more here