Improving Debugging For Optimized Rust Code

Master Thesis

Niklas Lundberg

Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology

September 15, 2021

Overview

1. Introduction
2. DWARF
3. Evaluation and Discussion
4. Conclusion

What is debugging

- Debugging is the process of finding an resolving errors, flaws, or faults.
- There are many different debugging techniques
- A debugger is a tool for debugging.
    - Used for controlling debugged target.
        - Start
        - Stop
        - Reset
        - Step
        - Breakpoints
    - Used to retrieve and visualize debug information.
        - Evaluate variable
        - Stack trace
        - And more

Unoptimized Vs Optimized Rust Code

- Rust is a programming language.
- Unoptimized Rust Code
  - Slow to run compared to other languages and optimized Rust code.
  - The machine code is very similar to the source code.
  - Values of variables are stored in memory on the stack.
  - Easy to debug.
- Optimized Rust Code
  - Fast compared to unoptimized code.
  - The machine code is very different form source code.
  - Values of variables have short life spans because they are stored in registers.
  - Hard to debug.

Problems

- Hard to debug because of short life spans of values.
- Existing debuggers like GDB often say that variables are optimized out.
- Existing debuggers also gives wrong result, this is especially true for LLDB.
- Both LLDB and GDB are hard to use for beginners.

Solution and motivation

- Research if better and more debug information can be generated.
    - Why? Because debuggers are restricted by the amount of debug information generated.
- Create a debugger for embedded systems that solve the mentioned problems.
    - Why? Because debuggers are very useful for embedded systems.
        - Small enough that the system can fully be understood.
        - They often have high requirements to meet.
        - Important that the optimized code work correctly.
    - Unoptimized code is to slow.
    - Written is Rust to use the benefits of Rust.

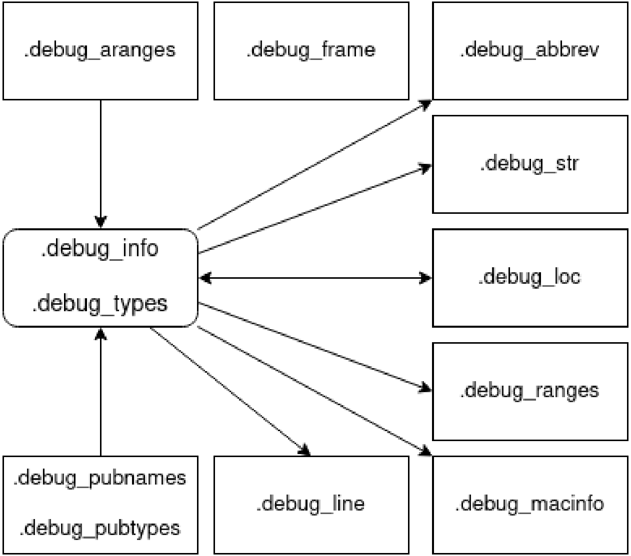
DWARF

DWARF

DWARF

- Debugging with Attributed Record Formats(DWARF)
- Debug information format
- Rust uses DWARF version 4
- DWARF is divided into 12 sections
- Executable and Linkable Format(ELF)

DWARF Sections



Debug Information Entry(DIE)

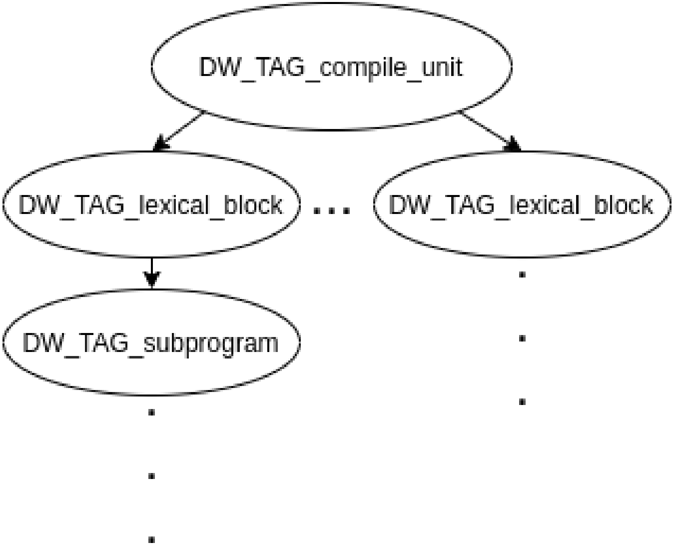
- Debug Information Entry(DIE).
- DWARF Attributes.
- DWARF DIE example from the .debug_info section.

```
<8><241>: Abbrev Number: 9 (DW_TAG_variable)
   <242>   DW_AT_location    : 2 byte block: 7d 3c     (DW_OP_breg13 (r13): 60)
   <245>   DW_AT_name        : (indirect string, offset: 0x40466): ptr
   <249>   DW_AT_decl_file   : 1
   <24a>   DW_AT_decl_line   : 591
   <24c>   DW_AT_type        : <0x1069>
```

Compilation unit

- Computer program is divided into compilation units.
- Each compilation unit contains a DIE tree.

Compilation unit



Evaluating a variable

- Find the current compilation unit.
- Find the current subprogram die.
- Find the searched variable die.
- Two parts to evaluating a variable:
    - Finding the location of the variable
    - Parsing the value into the correct type

Evaluating the location of a variable

```
<2><4321>: Abbrev Number: 16 (DW_TAG_subprogram)
   <4322>   DW_AT_low_pc       : 0x8000fca
   <4326>   DW_AT_high_pc      : 0x2c
   <432a>   DW_AT_frame_base   : 1 byte block: 57       (DW_OP_reg7 (r7))
   <432c>   DW_AT_linkage_name: (indirect string, offset: 0x473b8): _ZN24nucleo_r
   <4330>   DW_AT_name         : (indirect string, offset: 0x64a52): my_function
   <4334>   DW_AT_decl_file    : 1
   <4335>   DW_AT_decl_line    : 194
   <4336>   DW_AT_type         : <0x6233>
<3><433a>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <433b>   DW_AT_location     : 2 byte block: 91 7e    (DW_OP_fbreg: -2)
   <433e>   DW_AT_name         : (indirect string, offset: 0x11d94): val
   <4342>   DW_AT_decl_file    : 1
   <4343>   DW_AT_decl_line    : 194
   <4344>   DW_AT_type         : <0x6233>
```

Parsing the type of a variable

```
<1><6233>: Abbrev Number: 34 (DW_TAG_base_type)
   <6234>   DW_AT_name        : (indirect string, offset: 0x2a125): i16
   <6238>   DW_AT_encoding    : 5       (signed)
   <6239>   DW_AT_byte_size   : 2
```

Virtually Unwinding Call Stack

- Stack of subroutine activation's.
- A subroutine activation consists of:
  - Code location were the subroutine stopped
  - Preserved register values
  - Canonical Frame Address (CFA)
- The needed information is in section .debug_frame

Virtually Unwinding Subroutine Activation's

1. Find the Common Information Entry (CIE)
2. Find the Frame Description Entry (FDE)
3. Unwind CFA and register values.
4. Repeat for all activation's.

```
LOC CFA R0 R1 ... RN
L0
L1
...
LN
```

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_enum3 = TestEnum::Struct(TestStruct { flag: true, num: 123 });
```

ERD

```
test_enum3 = TestEnum { < OptimizedOut >}
```

GDB Version 11.0.90

```
(gdb) p test_enum3
$ 1 = nucleo_rtic_blinking_led::TestEnum::ITest(<optimized out>)
```

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_enum3 = TestEnum::Struct(TestStruct { flag: true, num: 123 });
```

LLDB Version 13.0.0

```
(nucleo_rtic_blinking_led::TestEnum) test_enum3 = {
  ITest = (0 = 0)
  UTest = (0 = 0)
  Struct = {
    0 = (flag = false, num = 0)
  }

  Non = {}
}
```

Comparing Evaluation Of Rust Enums

Rust Source Code

let mut test_struct = TestStruct { flag: true, num: 123 };

ERD

test_struct = TestStruct { num::123, flag::< OptimizedOut >}

GDB Version 11.0.90

(gdb) p test_struct
$ 1 = nucleo_rtic_blinking_led::TestStruct {flag: <sybthetic pointer>, num: 123}

LLDB Version 13.0.0

(nucleo_rtic_blinking_led::TestEnum) test_struct = (flag = false, num = 123)

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_u16: u16 = 500;
```

ERD

test_u16 = <OutOfRange>

GDB Version 11.0.90

```
(gdb) p test_u16
$ 1 = <optimized out>
```

LLDB Version 13.0.0

```
(unsigned short) test_u16 = <variable not available>
```

Conclusion

- Able to do some small improvements.
- ERD lacks some of the features that LLDB and GDB has.
- Contributed with a Debugging library for Rust.
- ERD is written in Rust.
- Still a lot that needs to be done.

Future Work

- Display last known value.
- Evaluating expressions in ERD.
- Display more information about the target system.

Demo

Thank you for listening