Port GDB to a New Architecture

齐尧
yao@codesourcery.com

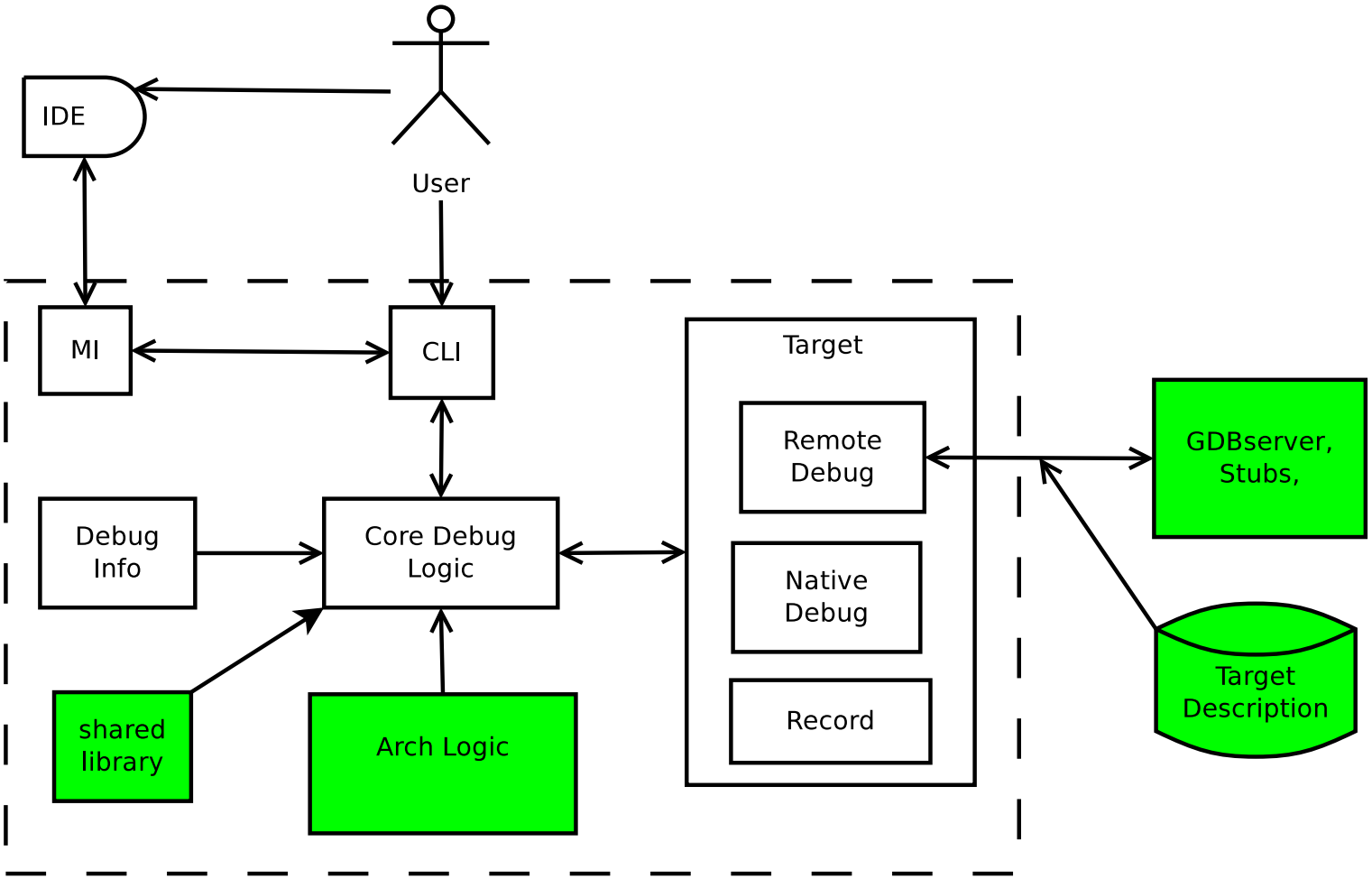
CodeSourcery/MentorGraphics

November 4, 2012

这个话题的目的

- 介绍GDB移植的一个基本步骤。虽然GDB已经被移植到多达20多种处理器上，但是缺少移植的步骤和细节。
- 介绍GDB的内部构造。通过移植的步骤，可以理解每个部件的功能，以及和其他部件的交互。
- 希望对其他正在做GDB移植或者试图理解GDB内部原理的工程师有所帮助。

What do we have to modify?



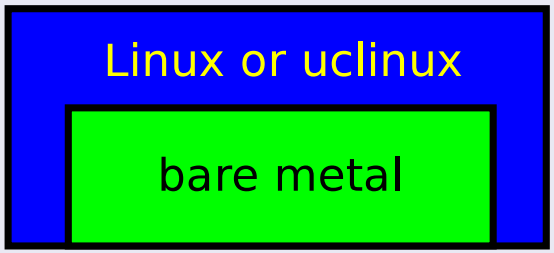
Two kinds of GDB

Bare metal or ELF

- breakpoint, software single step, disasembly,
- inferior call, `dwarf2_frame_init_reg`,
- prologue analyzer, frame unwinding by prologue, skip prologue,
- epilogue analysis/detection
- register type and group
- longjmp target

on Linux or ucLinux

- stub prologue analyzer,
- signal trampoline unwinding,
- next pc of syscall (rt_sigreturn, sigreturn)
- thread local storage



breakpoint & software singlestep

breakpoint

- A special instruction or an illegal instruction
- GDB 最基本的属性，移植的第一步

software single step

- 单步是GDB 的最基本属性，大多数处理器没有单步的硬件支持，需要软件模拟
- Insert breakpoint at the **next** instruction of pc
- Decode every instruction, and compute the **next** instruction of pc

inferior call

Widely used

- GDB 主动在target运行target函数
- `p func1()`
- allocate memory on target (call `malloc`)

GDB需要知道

- 函数的参数类型和返回类型
- 每个参数应该放在的位置 （寄存器 还是 stack）
- 返回值的位置 （寄存器 还是 stack）
- pass by value or pass by reference? `struct`
- alignment on passing stack
- `vararg`

inferior call in gdb

GDB hook	输入	作用
arch_push_dummy_call	type parameter and return. param value	根据ABI把参数放到应该放的地方
arch_return_value	返回值类型	根据ABI从正确位置取得返回值
arch_frame_align		frame alignment
arch_value_to_register **arch_register_to_value**		寄存器和Value之间的转换
arch_push_dummy_id		建立一个dummy frame id

prologue analyzer

prologue analyzer 干了两件事情

prologue 在哪里结束?	函数体的起始位置。
prologue 都干了写什么?	寄存器保存在哪里，特别是返回地址，`sp`, `fp` 保存在哪里

prologue analyzer 的作用

- 如果有**DWARF**信息，GDB会用**DWARF**进行 frame unwinding (主要是 `.debug_frame` section)
- 在没有**DWARF**信息的情况，GDB会分析函数的prologue，来进行frame unwinding。
- 为了正确解释**DWARF**的指令，GDB也需要知道函数体的起始位置。比如 `break foo`的时候。**arch**`_skip_prologue`。

frame unwinding by prologue

- 在知道了在prologue中，寄存器都保存在了什么位置，
- unwind frame的时候，用frame N的寄存器内容，去计算frame N+1 的寄存器内容

epilogue detection

- 当使用watchpoint监视一个local variable的时候，gdb需要注意scope。（如果是全局变量，不需要检查scope的）
- 当程序运行到epilogue的时候，这个时候没有办法知道epilogue 代码的scope，gdb就会错误的把watchpoint删除
- 通过 **arch_in_function_epilogue_p** 知道当前程序是否在epilogue。输入就是pc，通过分析指令，看是否是在epilogue。

```
1  static int
2  thumb_in_function_epilogue_p (struct gdbarch *gdbarch, CORE_ADDR pc)
3  {
4    /* inst1 is read from PC.  */
5    if ((inst1 & 0xff80) == 0x4700)
6      /* BX lr is typically used for returns.  */
7      found_return = 1;
8
9    if (!found_return)
10     return 0;
11
12   /* inst2 is read from PC - 2.  */
13   if ((inst2 & 0xff00) == 0xbc00)
14     /* POP (without PC).  */
15     found_stack_adjust = 1;
16
17   if (found_stack_adjust)
18     return 1;
19
20   return 0;
21 }
```

longjmp

- longjmp 应该跳回到setjmp 的位置，但是肯定不是一条指令实现的
- GDB需要知道它们之间的协议，正确的让程序停止在setjmp 的位置
- **arch**_get_longjmp_target

```
1  static int
2  foo_get_longjmp_target (struct frame_info *frame, CORE_ADDR *pc)
3  {
4    struct gdbarch *gdbarch = get_frame_arch (frame);
5    enum bfd_endian byte_order = gdbarch_byte_order (gdbarch);
6    CORE_ADDR jb_addr;
7    char buf[4];
8
9    /* JMP_BUF is passed by reference in R4.  */
10   jb_addr = get_frame_register_unsigned (frame, 4);
11
12   /* JMP_BUF contains 13 elements of type int, and return address is stored
13      in the last one.  */
14   if (target_read_memory (jb_addr + 12 * 4, buf, 4))
15     return 0;
16
17   *pc = extract_unsigned_integer (buf, 4, byte_order);
18
19   return 1;
20 }
```

Listing 2: foo_get_longjmp_target

stub prologue analyzer

Background

- 在调用共享库函数的时候需要有.plt .got的支持
- bl malloc@plt. Every time, in plt stub, code loads target address of malloc, and then jump to it.
- For the first time, dynamic linker is loaded to resolves the address of malloc, and update target address of malloc

GDB doesn't show details above

- 当用next命令，GDB会让程序运行到下一行，但是会跨过函数调用。
- 当程序在plt stub的时候，GDB必须知道当前程序在 inner frame。GDB不知道的话，就会让程序停下来。
- GDB需要有一个特殊的frame unwinder for stub

stub frame unwinder

```
1  static int
2  foo_stub_unwind_sniffer (const struct frame_unwind *self,
3                           struct frame_info *this_frame,
4                           void **this_prologue_cache)
5  {
6    CORE_ADDR addr_in_block;
7
8    addr_in_block = get_frame_address_in_block (this_frame);
9    if (in_plt_section (addr_in_block, NULL))
10     return 1;
11
12   return 0;
13 }
14
15 struct frame_unwind foo_stub_unwind = {
16   NORMAL_FRAME,
17   foo_stub_this_id,
18   foo_frame_prev_register,
19   NULL,
20   foo_stub_unwind_sniffer
21 };
```

Listing 3: stub frame unwinder

signal trampoline frame unwinding

- GDB已经有了很好的infrastructure对signal trampoline frame unwinding支持。
- 每个port需要定义和自己的signal trampoline匹配的instruction pattern。参见 `arm-linux-tdep.c:struct tramp_frame arm_linux_rt_sigreturn_tramp_frame`
- 而且还要从kernel的代码中看出，当在 signal trampoline frame的时候，`struct rt_sigframe` 所在位置。一般都是一个基于sp的偏移。参见 `arm-linux-tdep.c:arm_linux_sigreturn_init`

next pc of syscall

Next pc

- 在实现software single step的时候已经实现完了
- syscall的下一条指令地址不一定是 pc + 4，比如 sigreturn or rt_sigreturn。

GDB中实现

- 没有相应的GDB hook，但是MIPS和ARM有类似的实现。
- 在计算 next pc的函数中，考虑系统调用指令。查看系统调用的规范，知道系统调用号，如果是sigreturn or rt_sigreturn，从规范指定的寄存器中读出下一条指令的地址。