Port GDB to a new processor architecture:
TI C6x

Yao Qi
yao@codesourcery.com

CodeSourcery/MentorGraphics

2013-07-13
Outline

1. Overview
2. An overview of GDB
   - Two kinds of GDB
3. Bare metal or ELF
   - breakpoint & software single step
   - inferior call
   - frame unwinding
   - prologue analyzer
   - epilogue detection
   - longjmp
4. Linux or ucLinux
   - stub frame unwinder
   - signal trampoline frame unwinding
   - next pc of syscall
5. Conclusions

Yao Qi  yao@codesourcery.com  Port GDB to a new processor architecture: TI C6x
Introduction

- TMS320 C6000 series, or TMS320C6x: VLIW-based DSPs,
- Internal development in 2Q 2011, thank Bernd Schmidt and Andrew Jenner for the various explanations,
- Upstream from July to August 2011, thank Joseph Myers and GDB maintainers for the patient review,

Goals of this session

- List the **basic steps** of GDB porting. GDB has been ported to many different architectures, but the **steps** and **details** about porting is unclear.
- Make the GDB internals, **interactions** among different components, clear enough for GDB porting.
Typical submissions

[RFA 0/8] New port: TI C6x
[RFA 1/8] New port: TI C6x: Remove "gdb" from noconfigdirs in configure.ac
[RFA 2/8] New port: TI C6x: Handle tic6x-**linux and tic6x-** in configure.tgt
[RFA 3/8] New port: TI C6x: shared library for dsbt
[RFA 4/8] New port: TI C6x: Read loadmap from gdbserver
[RFA 5/8] New port: TI C6x: gdb port
[RFA 6/8] New port: TI C6x: gdbserver
[RFA 7/8] New port: TI C6x: test case fixes
[RFA 8/8] New port: TI C6x: NEWS

[PATCH] Add support for Tilera TILE-Gx processor (part 1/3: gdb)
[PATCH] Add support for Tilera TILE-Gx processor (part 2/3: gdb)
[PATCH] Add support for Tilera TILE-Gx processor (part 3/3: gdbserver)

[patch 0/4] Altera Nios II port
[patch 1/4] code changes for Nios II target
[patch 2/4] Nios II target descriptions
[patch 3/4] Nios II gdbserver support
[patch 4/4] Nios II testsuite fix
Steps matter!

In patch reader’s minds:

In patch writer’s minds:
## Two kinds of GDB

### Bare metal or ELF
- breakpoint, software single step,
- inferior call, `dwarf2_frame_init_reg`,
- prologue analyzer, frame unwinding by prologue, skip prologue,
- epilogue 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

---

Yao Qi  
yao@codesourcery.com

Port GDB to a new processor architecture: TI C6x
A special instruction or an illegal instruction,

The basic attribute of GDB, and the first step of porting,

```c
static const gdb_byte *
tic6x_breakpoint_from_pc (struct gdbarch *gdbarch, CORE_ADDR *bp_addr,
                           int *bp_size)
{
    struct gdbarch_tdep *tdep = gdbarch_tdep (gdbarch);
    *bp_size = 4;

    if (tdep == NULL || tdep->breakpoint == NULL)
        {
            if (BFD_ENDIAN_BIG == gdbarch_byte_order_for_code (gdbarch))
                return tic6x_bkpt_illegal_opcode_be;
            else
                return tic6x_bkpt_illegal_opcode_le;
        }
    else
        return tdep->breakpoint;
}
```

Listing 1: breakpoint
software single step

- Decode the instruction, and compute the next instruction of pc,
- Insert breakpoint at the next instruction of pc, resume and wait,

```c
static int
tic6x_software_single_step ( struct frame_info *frame )
{
    struct gdbarch *gdbarch = get_frame_arch (frame);
    struct address_space *aspace = get_frame_address_space (frame);
    CORE_ADDR next_pc = tic6x_get_next_pc (frame, get_frame_pc (frame));
    insert_single_step_breakpoint (gdbarch, aspace, next_pc);
    return 1;
}
```

Listing 2: software single step
breakpoint & software single step
inferior call
frame unwinding
prologue analyzer
epilogue detection
longjmp

Widely used

p func1(1, b, 10, 0.1, f)

GDB needs to know

- type of arguments and return,
- place to hold arguments (registers or stack),
- place of return value (registers or stack),
- pass by value or pass by reference?
- alignment of storing,
- vararg,

Yao Qi  yao@codesourcery.com

Port GDB to a new processor architecture:  TI C6x
inferior call in gdb

GDB will do the following steps:

1. Allocate a new piece of stack, which is called **dummy frame**, and put all arguments into the right place (registers and stack) and set the expected return address. Done by `arch_push_dummy_call`,

2. Adjust register `sp`. `arch_frame_align` is needed,

3. Push all the info needed to restore the caller’s state on dummy frame. `arch_dummy_id` is needed,

4. Set a breakpoint on the return address, and resume the program,

5. When breakpoint is hit, GDB gets the return value. Done by `arch_return_value`.
How GDB handles various different frames?

- Various unwinders are chained together,
- Iterate over the unwinders and associate a unwinder to the frame if the sniffer thinks this frame can be handled by the unwinder,
prologue analyzer

**Missions of prologue analyzer**

- Where is the end of prologue? starting address of function body,
- What does prologue do? where are registers saved? sp, fp?

**Functionality of prologue analyzer**

- frame unwinding by analyzing prologue,
- GDB needs to know the starting address of function body. When break foo, `arch_skip_prologue` is used to set the breakpoint at starting address of the function body.

Yao Qi  yao@codesourcery.com

Port GDB to a new processor architecture: TI C6x
prologue analyzer

static CORE_ADDR
tic6x_skip_prologue (struct gdbarch *gdbarch, CORE_ADDR start_pc)
{
    CORE_ADDR func_addr;
    struct tic6x_unwind_cache cache;

    /* See if we can determine the end of the prologue via the symbol table.
       If so, then return either PC, or the PC after the prologue, whichever is
       greater. */
    if (find_pc_partial_function (start_pc, NULL, &func_addr, NULL))
    {
        CORE_ADDR post_prologue_pc
            = skip_prologue_using_sal (gdbarch, func_addr);
        if (post_prologue_pc != 0)
            return max (start_pc, post_prologue_pc);
    }

    /* Can’t determine prologue from the symbol table, need to examine
       instructions. */
    return tic6x_analyze_prologue (gdbarch, start_pc, (CORE_ADDR) -1, &cache, NULL);
}

Listing 3: skip prologue
Suppose a watchpoint is used to monitor a local variable, gdb takes care of the scope of the local variable, When the program goes to epilogue, GDB will remove the watchpoint by mistake because it doesn’t know the program is still within the scope (in epilogue of the function),

```c
static int tic6x_in_function_epilogue_p (struct gdbarch *gdbarch, CORE_ADDR pc)
{
    unsigned long inst = tic6x_fetch_instruction (gdbarch, pc);
    /* Normally, the epilogue is composed by instruction ‘b .S2 b3’. */
    if ((inst & 0x0f83effc) == 0x360)
    {
        unsigned int src2 = tic6x_register_number ((inst >> 18) & 0x1f,
                                                   INST_S_BIT (inst),
                                                   INST_X_BIT (inst));

        if (src2 == TIC6X_RA_REGNUM)
            return 1;
    }
    return 0;
}
```

Yao Qi  yao@codesourcery.com  Port GDB to a new processor architecture: TI C6x
These fails are fixed:

FAIL: gdb.base/watch-cond.exp: watchpoint with local expression, local condition evaluates in correct frame
FAIL: gdb.mi/mi-watch.exp: sw: watchpoint trigger (unknown output after running)
FAIL: gdb.mi/mi2-watch.exp: sw: watchpoint trigger (unknown output after running)
longjmp

- GDB needs to know the **protocol** between longjmp and setjmp, and extract the target address of longjmp,
- so program can stop after the location of setjmp,

```c
static int tic6x_get_longjmp_target (struct frame_info *frame, CORE_ADDR *pc)
{
    struct gdbarch *gdbarch = get_frame_arch (frame);
    enum bfd_endian byte_order = gdbarch_byte_order (gdbarch);
    CORE_ADDR jb_addr;
    gdb_byte buf[4];

    /* JMP_BUF is passed by reference in A4. */
    jb_addr = get_frame_register_unsigned (frame, 4);

    /* JMP_BUF contains 13 elements of type int, and return address is stored in the last slot. */
    if (target_read_memory (jb_addr + 12 * 4, buf, 4))
        return 0;

    *pc = extract_unsigned_integer (buf, 4, byte_order);

    return 1;
}
```

Yao Qi  yao@codesourcery.com  Port GDB to a new processor architecture: TI C6x
These fails are fixed:

FAIL: gdb.base/longjmp.exp: next over longjmp(1)
FAIL: gdb.base/longjmp.exp: next over call_longjmp (2)
FAIL: gdb.base/longjmp.exp: next over patt3
Why does GDB need a stub frame unwinder

- When type command `next`, the program will cross the function call, and stop at the next line,
- GDB won’t stop the program if it is still in the inner frame,
- When the program is in plt stub, GDB can’t tell the program is still in the inner frame or not, so it stops the program by mistake,
- GDB needs a special frame unwinder for plt stub,
**stub frame unwinder**

Listing 6: stub frame unwinder

```c
static int tic6x_stub_unwind_sniffer (const struct frame_unwind *self,
                               struct frame_info *this_frame,
                               void **this_prologue_cache)
{
    CORE_ADDR addr_in_block;
    addr_in_block = get_frame_address_in_block (this_frame);
    if (in_plt_section (addr_in_block))
        return 1;
    return 0;
}

static const struct frame_unwind tic6x_stub_unwind =
{
    NORMAL_FRAME,
    default_frame_unwind_stop_reason,
    tic6x_stub_this_id,
    tic6x_frame_prev_register,
    NULL,
    tic6x_stub_unwind_sniffer
};
```

Yao Qi  yao@codesourcery.com
Port GDB to a new processor architecture:  TI C6x
signal trampoline frame unwinding

- GDB has a good infrastructure to do frame unwinding in signal trampoline,
- Each port has to define its instruction pattern to match its signal trampoline code sequence.

```c
static struct tramp_frame tic6x_linux_rt_sigreturn_tramp_frame =
{
    SIGTRAMP_FRAME,
    4,
    {
        {0x000045aa, 0xffffffff}, /* mvk .S2 139,b0 */
        {0x10000000, -1}, /* swe */
        {TRAMP_SENTINEL_INSN}
    },
    tic6x_linux_rt_sigreturn_init
};
```

Listing 7: tic6x_linux_rt__sigreturn_tramp_frame
Check the kernel source and find the location of struct `rt_sigframe` on stack. Usually, it is an offset based on `sp`.

```c
static void
tic6x_linux_rt_sigreturn_init (const struct tramp_frame *self,
                               struct frame_info *this_frame,
                               struct trad_frame_cache *this_cache,
                               CORE_ADDR func)
{
    struct gdbarch *gdbarch = get_frame_arch (this_frame);
    CORE_ADDR sp = get_frame_register_unsigned (this_frame, TIC6X_SP_REGNUM);
    /* The base of struct sigcontext is computed by examining the definition of
     * struct rt_sigframe in linux kernel source arch/c6x/kernel/signal.c. */
    CORE_ADDR base = (sp + TIC6X_SP_RT_SIGFRAME
        + 4 + 4
        + TIC6X_SIGINFO_SIZE
        + 4 + 4 /* uc_flags and *uc_link in struct ucontext. */
        + TIC6X_STACK_T_SIZE);
```

Listing 8: tic6x_linux_rt__sigreturn_init
next pc of syscall

Next pc

- GDB backend on software single step computes the next pc of a given instruction,
- except for syscall, such as \texttt{sigreturn} or \texttt{rt\_sigreturn}.

Implementation in GDB

- There are similar implementations in MIPS, ARM, C6x and NIOS 2,
- When computing the next pc, take syscall into account. Get the syscall number, if it is \texttt{sigreturn} or \texttt{rt\_sigreturn}, get the address from a certain register specified by the specification.
next pc of syscall

Listing 9: next pc of syscall
next pc of syscall

```c
static CORE_ADDR
tic6x_get_next_pc (struct frame_info *frame, CORE_ADDR pc)
{
    struct gdbarch *gdbarch = get_frame_arch (frame);
    unsigned long inst;
    int register_number;
    int last = 0;

    do {
        inst = tic6x_fetch_instruction (gdbarch, pc);
        last = !(inst & 1);
        if (inst == TIC6X_INST_SWE) {
            struct gdbarch_tdep *tdep = gdbarch_tdep (gdbarch);
            if (tdep->syscall_next_pc != NULL) {
                return tdep->syscall_next_pc (frame);
            }
        } else if (inst == TIC6X_INST_SYSCALL) {
            CORE_ADDR pc = tic6x_get_next_pc (frame);
            if (pc != NULL) {
                return pc;
            }
        }
    }
    return NULL;
}
```

Listing 10: get next pc
Conclusions

Follow the steps during the porting,

GDB porting is a process:
FAIL -> component -> patch -> PASS

Track the changes of test results after a feature is ported,
Be familiar with GDB testsuite,
Have a look at other ports,
Fix as many fails as you can,

Questions?

Yao Qi  yao@codesourcery.com
What do we have to modify?

IDE

User

MI

CLI

Core Debug Logic

Debug Info

shared library

Arch Logic

Target

Remote Debug

Native Debug

Record

GDBserver, Stubs,

Target Description