A Symbolic Debugger for PowerPC-Based Hardware, Using the Engineering Support Processor (ESP)

Kiyokuni KAWACHIYA and Takao MORIYAMA

IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242, Japan
<{kawatiya,moriyama}@jp.ibm.com>

Abstract

For debugging PowerPC-based hardware systems, there is a tool named the Engineering Support Processor (ESP) that accesses and controls the chip via the JTAG interface. With the ESP, a user can debug a target system by starting and stopping it, accessing registers and memory, and so on. However, with ESP alone, it is difficult to symbolically debug programs written in high-level languages such as C. Therefore, we have developed a mechanism for using the GNU debugger (GDB), which is a widely-used symbolic debugger, through the ESP. This mechanism is implemented by a program named GDBserver, which mediates between a host-GDB and the ESP. This report describes its structure and detailed implementation.

Keywords: Symbolic Debugger, Development Environment, PowerPC, GDB Server

1 Introduction

The Engineering Support Processor (ESP) [1] is a debugging tool for hardware systems using PowerPC chips [2]. It consists of a hardware part with an interface to a JTAG (IEEE 1149.1) port and buffers, and a software part for AIX. By using the ESP, a user can debug a target system by accessing PowerPC registers (including special registers) and memory, downloading, starting, and stopping a program, setting breakpoints, and so on. In addition to a GUI-based user interface on X Window System, the ESP provides a programming interface named EZsockets, which allows the user to control the ESP from another program via a network [3, 4].

With the ESP, a program on a target system can be debugged at the machine-language level. However, it is difficult to symbolically debug programs written in high-level languages such as C. For this purpose, some symbolic debugger such as the GNU debugger (GDB) [5] is necessary. The GDB already supports the PowerPC as a target CPU, and can debug a program for the PowerPC symbolically. However, it may be somewhat difficult to run the GDB itself on an unstable target system.

For debugging programs in the hardware development phase, the GDB provides a remote-debugging facility, which makes it possible to debug a program on a target system via a serial line or network. A unique GDB remote serial protocol[1] is used for this remote debugging.

[1]An overview of this protocol is given as a set of comments in a GDB source file, `remote.c`.

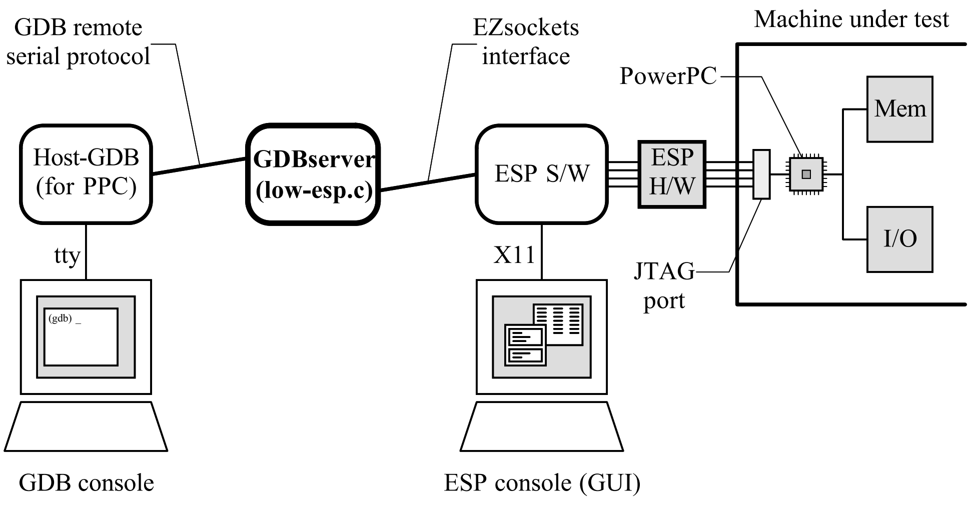

Figure 1: Debugging environment using the GDBserver for ESP

We have developed a program named "GDBserver for ESP" that mediates between this remote-debugging function and the EZsockets interface of the ESP. This approach allows the GDB to run on a stable host system and to debug an (unstable) target system symbolically through the ESP. The present report describes the structure and detailed implementation of the debugging environment. The basic structure is first described in Section 2, and several special considerations are discussed in Section 3. Section 4 discusses several points to be noted in using the debugging environment, and Section 5 summarizes our work. Topics that concern specific target systems are dealt with separately in another report, RT5131 [6].

2 Basic Structure

Figure 1 shows the structure of the debugging environment using the GDBserver for ESP. The host-GDB's debugging directives through the GDB remote serial protocol are translated into a series of ESP commands by the GDBserver. The translated commands are sent to the ESP through the EZsockets interface and executed by the ESP. In parallel with the commands from the GDBserver, the ESP can be controlled through its original GUI. Through this "back door," it is possible to control special registers and so on that are not supported in the GDB.

The remainder of this section concerns functions of these modules.

2.1 Host-GDB and Upper Part of GDBserver

The host-GDB is a regular GDB built for the PowerPC. The version that we used to develop the GDBserver was GDB version 4.16. `powerpc-ibm-aix3.2.5` was specified as its configuration option.[2] We tried to change the GDB itself as little as possible, so that we can follow up future improvements easily. The only change we made was to modify `Makefile.in` by incorporating a file named `ser-tcp.c` to allow remote debugging via a TCP/IP network.

[2]AIX 3.2.5 is specified, to match the working environment of the ESP.

Table 1: Functions provided by the lower part of the GDBserver (`low-esp.c`)

Name	Description
`create_inferior()`	Start the target program
`kill_inferior()`	Kill the target program
`mythread_alive()`	Check whether the target program is alive
`interrupt_inferior()`	Interrupt and stop the target program
`mywait()`	Wait for the target program to stop
`myresume()`	Resume the target program
`registers[]`	Area for storing registers while stopped
`fetch_inferior_registers()`	Read specified register(s)
`store_inferior_registers()`	Write specified register(s)
`read_inferior_memory()`	Read specified memory area
`write_inferior_memory()`	Write specified memory area

The GDBserver receives directives through the GDB remote serial protocol from the host-GDB, which is connected via a network,[3] and translates them into ESP commands. Directives to be handled include starting and stopping a target program, waiting for the program to stop, and reading and writing registers and memory. The GDBserver for ESP was developed by linking a GDBserver skeleton included in the GDB package with a lower-part module for accessing the ESP. In the current implementation, the upper-part module, which mainly handles the protocol (`server.c`, `remote-utils.c`, etc.), is used almost as provided except for the addition of invocation options and changes to the interrupt code and register access timing.

2.2 Lower Part of GDBserver

The lower part of the GDBserver (`low-esp.c`), which is linked to the upper part described in the previous subsection, is the central part of the implementation. Functions to be provided here include starting and stopping the target program, and reading and writing registers and memory. The complete list is shown in Table 1. Figure 2 illustrates the states of a target program and situations in which each function is called. Most functions are called when the target program has been stopped.

The purpose of each function is as follows:

`create_inferior()` is a function for starting the target program, and is called when the GDB-server is invoked. First, it creates an EZsockets connection and checks the target system.[4] Next, it downloads and invokes the target program by using ESP commands. Since an ESP's hardware breakpoint is set before the invocation, the target program is stopped at its entry point.

In our implementation, the GDBserver can be attached to an already-running target program. When this feature is specified in the invocation option of the GDBserver,

[3]The connection is based on TCP/IP, and it is possible that the host-GDB and GDBserver reside on the same machine.

[4]This function is also called when the target program is restarted by using GDB's `run` command. In this case, the connecting and checking are skipped.

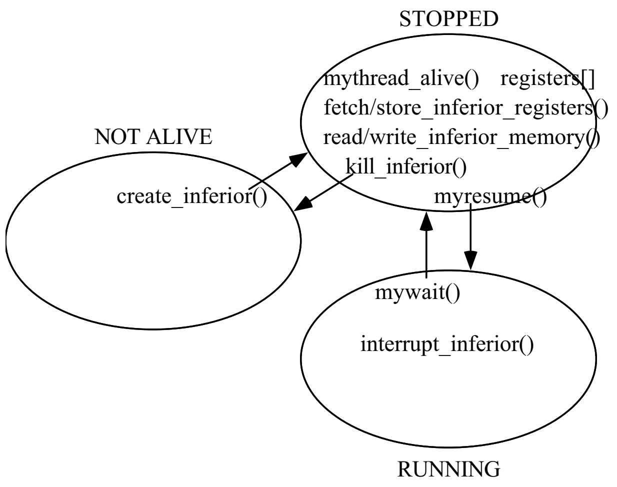

Figure 2: States of the target program

`create_inferior()` stops the target program instead of downloading and invoking a new target program.

`kill_inferior()` is a function for killing the target program. In the current implementation, it only changes the internal state of the GDBserver, and issues no ESP command.

`mythread_alive()` is a function for checking whether the target program is still available (alive), and returns 0 or 1 according to the above-mentioned internal state.

`interrupt_inferior()` is a function that we have added to the original GDBserver skeleton. It is called in a signal-processing context when an interrupt comes from the host-GDB. In this function, only a flag to indicate the interrupt is set, and the actual processing to stop the target program is done in the next `mywait()`.

`mywait()` is a function for waiting until the target program is stopped. There are two possible reasons for a program to be stopped: a forced stop by one of the above-mentioned interrupts, and a software breakpoint specified by a user from the host-GDB. As reason codes returned to the GDB, `SIGINT` and `SIGTRAP` are used, respectively. In the current implementation, stopping with other exceptions (such as division by zero) is not supported.

`myresume()` is a function for resuming the stopped target program.

`registers[]` is an area for storing registers used by the following two functions. Information stored here is also accessed directly from the upper part of the GDBserver.

`fetch_inferior_registers()` **and** `store_inferior_registers()` are functions for reading or writing specified PowerPC register(s) from or into the `registers[]`. These functions can be called only while the target program is stopped.
The GDB that we used tries to access the MQ register as one of the user registers even when the PowerPC is specified as its configuration. Because there is no MQ register in the PowerPC, the FPSCR register is accessed instead in our implementation.

`read_inferior_memory()` **and** `write_inferior_memory()` are functions for reading from or writing to memory on the target system. These functions are also called only while the target program is stopped. The ESP can access memory only by its physical address. Therefore, simple address translations to satisfy the target program are performed within these

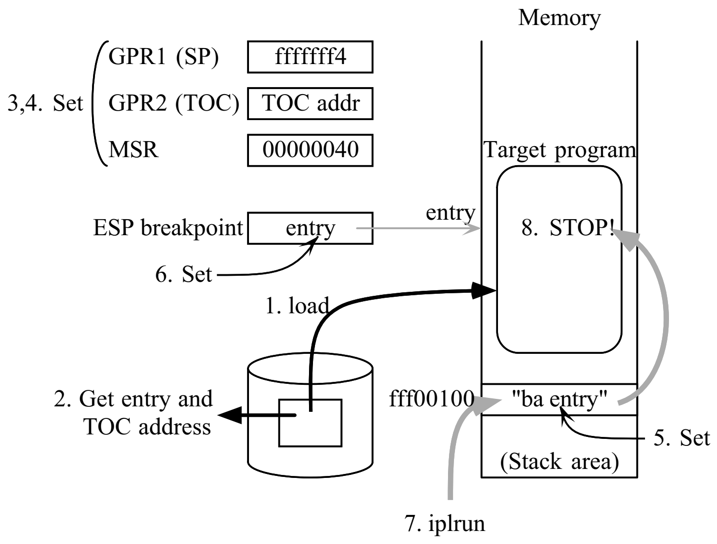

Figure 3: Starting the target program

functions.

3 Special Considerations

This section describes several points that required special consideration in implementation.

3.1 Starting the Target Program

The target program is started by `create_inferior()` through the following steps (Figure 3):

1. Download the target program by using ESP's `load` command. This command reads each section into the memory based on address information in the XCOFF header.
2. Analyze the XCOFF header and get the addresses of the entry point and TOC (table of contents).
3. Initialize GPR1 (the stack pointer)[5] and GPR2 (the TOC register).
4. Set `0x00000040` to MSR. When this is done, the jump address at the resetting in Step 7 becomes `0xfff00100`.
5. Create a branch instruction to the entry address (`ba entry`) and write it to the reset vector address `0xfff00100`.
6. Set the ESP's hardware breakpoint at the entry address.
7. Reset the PowerPC by using ESP's `iplrun` (reset the target) command.
8. The target program is stopped at its entry point by the hardware breakpoint set in Step 6.

Through this procedure, the target program is downloaded and stopped at its first instruction. In the current implementation, passing of arguments to the target program is not supported.

[5]The initial value of the stack pointer depends on the target system, and can be specified by using an invocation option of the GDBserver. In the current implementation, `0xfffffff4` is used as a default initial value of the stack pointer. This is because three words higher than the stack pointer may be used by the called function in the standard linkage convention of the PowerPC [7].

3.2 Making the Target Program Resume

`myresume()` makes a stopped target program resume through the following steps:

1. Set the ESP's hardware breakpoint at a program-exception vector address `0xfff00700`.
2. Make the target program resume by using ESP's `run 0` (run the target) command.

In the GDB, breakpoints are implemented by embedding a trap instruction[6] into the target program. When a trap occurs, the PowerPC's execution jumps to the program-exception vector address `0xfff00700`. The target program is stopped by the ESP's hardware breakpoint set in Step 1. The argument `0` of the `run` command is specified to allow the ESP operation to be performed while the target system is running.

3.3 Waiting for the Target Program to Stop

Until the target program is stopped, the GDBserver blocks in `mywait()`. There are two possible reasons for the target program to be stopped: the trap instruction mentioned above, and an interrupt from the host-GDB (usually caused by Ctrl-C). When the GDBserver is notified of an interrupt through an asynchronous I/O signal, `interrupt_inferior()` is called. This function merely sets a flag to indicate the interrupt, and the actual processing needed to stop the program is done in `mywait()` by checking this flag.

The processing in `mywait()` is as follows:

1. Execute the following loop until the target program stops.
   a. If the interrupted flag is set by `interrupt_inferior()`, stop the target program by using ESP's `stop` (stop the target) command.
   b. Check the status of the target program by using ESP's `wait -v` (check status) command. If the program has been stopped, exit the loop and go to Step 2.
   c. Sleep for 1 second.[7]
   d. Return to Step 1a.
2. Check IAR (the program counter). If the target program is stopped in the trap handler (`0xfff00700`), make the program resume as follows until it exits the handler:
   a. Set the ESP's hardware breakpoint at the trapped address (saved in SRR0).
   b. Write `0x4c000064` (`rfi` instruction) to `0xfff00700`, and resume the program by using ESP's `run` (run the target) command.
   c. The target program is stopped again at the trapped address by the hardware breakpoint set in Step 2a.
3. If the processor's caches are turned on, flush them. This procedure will be described in detail in the next subsection.
4. Set the reason for the stop (`SIGINT` or `SIGTRAP`) and return.

Figure 4 shows a sequence in which a program is stopped by a trap. Since the trap handler is exited in Step 2, the chip status can be directly used as the status of registers and so on. This simplification is also useful for avoiding confusion in direct debugging from the ESP.

[6] Precisely speaking, an unconditional trap instruction `0x7d821008` (`twge r2,r2`) is used.

[7] This sleep causes a delay of at most about 1 second for an interruption by Ctrl-C. If this delay becomes a problem, it is possible to issue ESP's `stop` command directly from its native GUI.

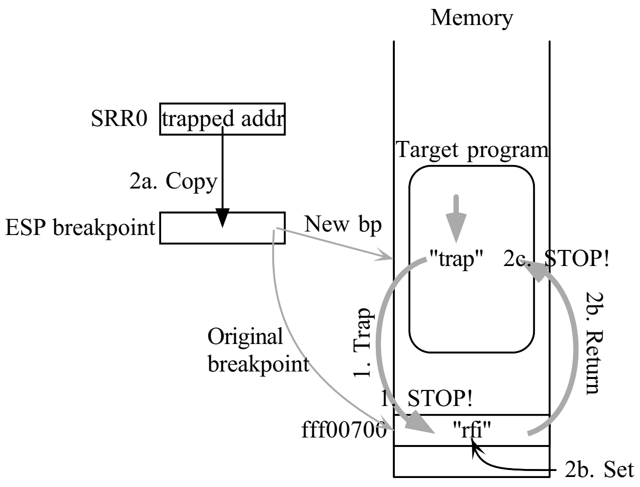

Figure 4: Stopping the target program by means of a trap

3.4 Flushing Caches after Stopping

In accessing memories, the ESP bypasses data and instruction caches in the CPU. Therefore, to handle memory-access requests from the host-GDB correctly, the caches must be flushed (written back and invalidated) in advance. Since the ESP cannot flush caches, a program for flushing caches is executed by the PowerPC itself when caches are being used. This is done every time the target program stops (in Step 3 of the process described in the previous subsection).

This program flushes data-cache blocks (`dcbf` instruction) and invalidates instruction-cache blocks (`icbi` instruction)[8] successively for the whole physical memory loaded on the target system. The memory area of the program uses about 10–20 bytes from `0xfff00710`, which is an area for the trap handler.

3.5 Memory Address Translation

Memory is accessed from the ESP according to its physical address. Therefore, when address translation is used in the target program, the same translation must be done by software in the GDBserver. This translation is necessary in `read_inferior_memory()` and `write_inferior_memory()`. The details of the translation depend on the target system and the target program. But it can be presumed that in most target systems requiring the debugging environment described here, address translation is not used, or is used only for a small fixed area to map I/O spaces, and so on. Therefore, the current implementation deals with the address translation problem by registering such static fixed translations in advance.

It may be possible to deal with the problem in earnest by reading BAT registers and page tables and simulating the address translation by means of software in the GDBserver. However, this technique is not adopted in the current implementation because of its large processing overhead.

[8]In the case of the PowerPC 604 [8], the instruction cache can be invalidated by setting bit 20 of the HID0 register, and this is used.

3.6 Performance Tuning

Access to the ESP through EZsockets is not very fast. Therefore, requests to the ESP should be aggregated for portions that are executed often.

In the current implementation of the GDB remote serial protocol, all user-level registers are fetched on a register access, and it is important to tune the register-fetching code. Therefore, in reading all registers by means of the function `fetch_inferior_registers()`, a macro program for the ESP that reads all registers and returns the collected result is executed instead of sending ESP's register-fetch commands successively.

Such read-aggregation tuning is also applied to `read_inferior_memory()`. This is important when displaying the contents of large arrays and so on. However, a macro program is not used in this case; instead, ESP's continuous memory read command (`memread`) is used.

4 About the Target Program and Debugging Environment

This section describes general points to be noted when using the debugging environment. Actual use of the GDBserver, which depends on the hardware structure of the target system and so on, will be described in another report [6].

4.1 Compiling a Target Program

Information for symbolic debugging must be included in the object. In most C compilers, this can be indicated by the `-g` option. In the ESP and the GDBserver, in the current implementation, address relocation on downloading of the program is not supported. Therefore, the load addresses should be explicitly indicated in the XCOFF header of the object. A typical method of compiling the program to take account of these points is as follows:[9]

```
    % cc -c -g targetprog.c
    % ld -T0xff800000 -D0xff810000 targetprog.o -e entry -o targetprog
```

The built object must run in a stand-alone environment, and must not use system calls or dynamic-link libraries; the latter point can be checked by using the `dump -H` command in AIX.

4.2 Writing a Target Program

In writing a target program, the following points should be taken into account:

- The bss section is not cleared on loading. Therefore, the initial value of a global variable that is not explicitly initialized is unpredictable.
- Arguments cannot be passed at invocation.
- The program must not "exit," since its behavior when it does so is unpredictable.
- The program must not cause exceptions, unless it provides its own exception handlers.
- The trap instruction cannot be used, because it is used by the GDB.
- The area for the program-exception handler (`0xfff00700`–`7ff`) is overwritten by the GDBserver whenever necessary, although the program can store some data there.
- MSR[IP] (bit 25) must not be cleared, since doing so changes the exception vector addresses, and causes the trap to be handled incorrectly.

[9]The options of `ld` are for AIX 3.2.5.

- The address translation can be used only if it is a static translation registered in the GDBserver in advance. Basically, it is preferable that the effective address should be the same as the physical address (V=R).
- The IABR register[10] must not be modified, because it is used by the ESP for implementing the hardware breakpoint.

4.3 Restrictions

Finally, the debugging environment has the following restrictions:

- GDB was originally developed for debugging user-level programs. Therefore, the supported registers are limited to user-level registers.[11] Supervisor-level registers such as BAT and SRR0/1 cannot be handled by the GDB. However, it is possible to control these registers directly from the ESP when the program is stopped.
- It is not possible to resume with specifying a signal (`signal` command).
- Some commands cannot be used in practice because their speed is too slow. For example, one-line execution of the source code (`next` command) is slow because it is implemented by repeating single-step execution and IAR-register checking until the first instruction of the next line is reached.
- When ESP's various windows (the register-displaying window, etc.) are displayed in auto-update mode, the processing becomes very slow because of the frequency of update intervention.

5 Summary

We have described an environment that allows symbolic debugging of PowerPC-based hardware systems. This debugging environment is implemented by a program named GDBserver, which mediates between the host-GDB and the ESP, a device for accessing the chip through its JTAG interface. The debugging directives from the host-GDB through the GDB remote serial protocol are translated into a series of ESP commands by the GDBserver. This approach enables the debugging environment to run on a stable host system and to symbolically debug the target system through the ESP.

Acknowledgment

The authors would like to thank to Kent D. Thompson of IBM Austin, who provided valuable advice on using and programming the ESP.

References

[1] K. D. Thompson and J. Bordovsky: *Engineering Support Processor (ESP) 604 User's Reference Manual*, IBM Austin, Texas (1996).

[2] IBM: *The PowerPC Architecture*, Morgan Kaufmann Publishers, Inc. (1994).

[10]This register is unique to the PowerPC 604.

[11]Strictly speaking, 71 registers of GPR0–31, FPR0–31, IAR, MSR, CR, LR, CTR, XER, and FPSCR.

[3] K. D. Thompson and J. Bordovsky: *Engineering Support Processor (ESP) Training Manual*, IBM Austin, Texas (1996).

[4] K. D. Thompson and J. Bordovsky: *Engineering Support Processor (ESP) Programmers Notes*, IBM Austin, Texas (1996).

[5] R. M. Stallman and Cygnus Support: *Debugging with GDB version 4.16*, Free Software Foundation (1994).

[6] K. Kawachiya and T. Moriyama: *Implementation Details of the GDBserver for the ESP/MRCJ System*, IBM Research Report, RT5131, IBM (1997).

[7] IBM: *AIX Version 3.2 Assembler Language Reference*, Chapter 5, IBM Manual SC23-2197-02.

[8] IBM Microelectronics and Motorola: *PowerPC 604 RISC Microprocessor User's Manual.*

August 27, 1997

RT0212

Computer Science 10 pages

Research Report

A Symbolic Debugger for PowerPC-Based Hardware, Using the Engineering Support Processor (ESP)

Kiyokuni KAWACHIYA and Takao MORIYAMA

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242, Japan