Automatic Generation of GNU Binutils and GDB for Custom Processors Based on Plug-in Method

Takahiro Kumura [1, 2] Soichiro Taga [3] Nagisa Ishiura [4]
Yoshinori Takeuchi [2] Masaharu Imai [2]

[1] System IP Core Research Laboratories, NEC Corporation
[2] Graduate School of Information Science and Technology, Osaka University
[3] Mitsubishi Electric Micro-Computer Application Software Co., Ltd.
[4] School of Science and Technology, Kwansei Gakuin University

Abstract— This paper presents a scheme of auto-generating GNU software development tools for newly developed processor cores based on a plug-in method. The plug-in method was originally designed to generate toolchains for configurable processors by augmenting base processor's existing tools using plug-ins, i.e. components of the tools that handle extra instructions and registers. This paper shows that the plug-in method can be applied to generating tools for arbitrary instruction set architectures by adding instructions and registers to an empty base processor with no instructions nor registers. An experimental system based on our method successfully generated a GNU toolchain consisting of an assembler, a disassembler, a linker, a simulator, and a debugger from succinct architecture description. Although the generated GDB supports only assembly level debugging, this is the first system that retargets the GDB debugger automatically.

I. Introduction

Processor cores are the central components for constructing SoCs which may dominate overall performance, cost, and power consumption of the systems. Among the variety of choices for the processor cores are custom processors or ASIPs (application domain specific instruction set processors). With a high performance-to-power ratio brought about by instruction sets and micro-architectures optimized for target application domains, they are utilized for various applications such as multimedia processing, wireless communications, and medical and health cares [1], [2]. Tools to support ASIP design such as [3] and [4] are available now.

In order to promote ASIP cores, however, it is very important to expedite development of software development tools, as well as the design of the processors. This is especially true in the recent SoC development circumstances, where performance estimation and architecture exploration must be done in the early design stage, and thus it is desirable to have the software development tools as early as the specification of the processors is drawn up. In this view, there has been a lot of attempts to generate (auto-retarget) software development tools for given instruction set architectures (ISAs).

One major category of the tool generation systems [5], [6], [7], [8] use their own frameworks of assemblers, linkers, simulators, etc., where the usage and customization of the generated tools may be strictly controlled by their licenses. Another category of systems [9], [10], [11] work on the GNU [12] toolchain, which is allowed to redistribute or to modify under the GPL [13]. However, to the best of authors' knowledge, the existing retargeting systems can generate Binutils (GNU binary utilities which include an assembler, a linker, etc.) and instruction set simulators (ISSs), but not the GDB debugger. This is because retargeting of the GDB debugger involves many tasks such as handling breakpoints on the ISS and analyzing stack frames based on procedure calling conventions, and it is difficult to automate all of them.

This paper attempts to solve a part of this problem, specifically, handling breakpoints, by applying the plug-in method proposed in [14] to auto-retargeting GNU software development tools. A plug-in is a software component that handles extra instructions and registers in each tool, which is used to augment the functionality of the existing tools in [14]. We will retarget the GNU toolchain to arbitrary processor architectures by extending *framework tools*, which consist only of the machine independent portion of the tools, using plug-ins that includes flexible instruction parser, flexible relocation resolver, etc. This scheme provides a comprehensible view in retargeting GNU Binutils and GDB. An assembler, a disassembler, and a linker in Binutils and an ISS in GDB will be easily auto-retargeted by our scheme. As for debugging part of GDB, we concentrate only on the components necessary for assembly level debugging. Since the plug-ins for GDB in [14] covers only simulation and display of register contents, we enhance them so as to handle breakpoints.

We implemented a GNU toolchain retargeting system based on the proposed method and applied it to a 16-bit processor MeDIX-I [15] with 48 instructions, which was developed for ultra low power SoCs for medical applications, and a 32 bit in-house RISC processor with 461 instructions. From ISA specification of 1,475 and 18,586 lines, respectively, Binutils and GDB are generated in which the functionality of the assemblers, the disassemblers, the linkers, the simulators, as well as the debuggers are successfully implemented.

II. Related Work

There are basically two ways of generating software development tools: one is to extend an existing toolchain, and the other is to retarget (to rewrite machine dependent potion of) an existing tool chain.

The former scheme is suited for generating a toolchain for configurable processors which are designed by augmenting some base processor whose toolchain is already available. The tool generation system for Tensilica's Xtensa [4], [16] delivers GCC, as well as Binutils, GDB, and ISS for target processors which are configured by adding user defined instructions, extra registers, etc. to the Xtensa base processor. However, the tool generator supports only Xtensa, and there is no literature regarding user defined relocation type, as far as the authors know. On the other hand, a tool extension system based on a plug-in method [14] is more flexible so that it is applicable to multiple base processors and it supports user defined relocation types. However, in any case, this class of toolchain generators are based on the premise that target processors are derived from some base processor.

The latter scheme accepts broader classes of target ISAs. ACE's Cosy [7] and Coware's LISATek [8] generate software development environment by retargeting their original toolchains. Other systems presented in [5] and [6] are also based on their own tools. Although they can generate sophisticated tools including compilers, redistribution and customization of generated tools are strongly controlled by their licenses.

In contrast, the GNU toolchain is allowed to redistribute or to modify under the GPL [13]. Major efforts to auto-retarget GNU toolchain includes CGEN [9], rbinutils [10], and ArchC [11]. RedHat's CGEN generates the opcode library, which performs encoding/decoding of instructions, and the functions for simulating instruction execution so as to retarget Binutils and ISS, but not the GDB debugger. Rbinutils generates Binutils from ISA specification, but not GDB nor ISS. ArchC is an open software that generates Binutils and ISS from architectural specification. It was reported in [11] that Binutils and ISS for multiple ISAs were generated, but the GDB debugger has not been retargeted. This is because retargeting of debugger requires many functions to be tailored for the target, which include display of register and memory contents, handling of breakpoints, and stack frame analysis, and it is difficult to automate all of them.

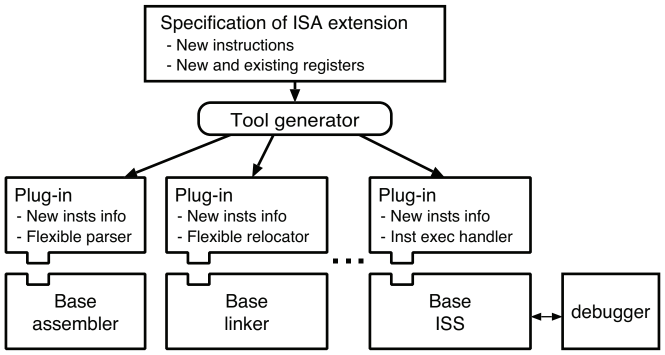

Fig. 1. Concept of the plug-in method.

III. Plug-in Method [14]

The plug-in method was originally developed for the purpose of generating software development tools for configurable processors, whose functionality was extended by adding new instructions and registers to some base processors. Fig. 1 illustrates the concept of the plug-in method [14]. It is assumed that tools such as an assembler, a linker, and an ISS for the base processor are available. A plug-in is a software component which performs the tasks regarding newly added instructions and registers. The base tools are modified so that they can call the corresponding plug-ins when they encounter the new features. The plug-ins are generated by "Tool generator" from specification of ISA extension, so the tools for the augmented ISA are obtained automatically.

The input to the tool generator consists of 1) XML document that describes the extended part of the ISA and 2) behavior specification that defines the behavior of the newly added instructions. Fig. 2 is an example of the XML document. Lines 10–27 are the specification of a newly added instruction whose mnemonic, assembly syntax, field information, and the attribute of the operands are declared. Lines 3–8 defines the registers where `GPR` (lines 4–5) is a preexisting register file (attribute `base="true"`) while `MYGPR` (lines 6–7) is a new register file (`base="false"`). The behavior of the added instruction is written in the C syntax, as shown in Fig. 3, which will be incorporated into the ISS.

IV. Automatic Retargeting of Binutils and GDB

A. Overview of the Proposed Method

We enhance the plug-in method and apply it to auto-retargeting GNU tools for processors having whole new ISAs. Fig. 4 illustrates the flow of tool generation in our

```
<Processor>
<nickname>cpu</nickname>
<register_type length="32">GPR_type</register_type>
<register_bank type="GPR_type" size="32" prefix="R"
   base="true">GPR</register_bank>
<register_bank type="GPR_type" size="8" prefix="MR"
   base="false">MYGPR</register_bank>
<insn_length>32</insn_length>

<insn>
   <mnemonic>MYADD</mnemonic>
   <syntax>MYADD %reg1, %reg2, %reg3</syntax>
   <field type="GPR" length="5">reg1</field>
   <field type="opcode" value="0b111111"
      length="6">opc0</field>
   <field type="GPR" length="5">reg2</field>
   <field type="opcode" value="0b1111001000"
      length="11">opc1</field>
   <field type="GPR" length="5">reg3</field>
   <input>
      <operand type="GPR" width="32">reg1</operand>
      <operand type="GPR" width="32">reg2</operand>
   </input>
   <output>
      <operand type="GPR" width="32">reg3</operand>
   <output>
</insn>

<behavior>cpu-isa.c</behavior>
</Processor>
```

Fig. 2. Specification of instruction set extension.

```
behavior (MYADD)
{
   int32_t val_reg1 = REG_read32 (GPR, reg1);
   int32_t val_reg2 = REG_read32 (GPR, reg2);
   int32_t val_reg3;
   val_reg3 = val_reg1 + val_reg2;
   REG_write32 (GPR, reg3, val_reg3);
}
```

Fig. 3. Behavior specification of an instruction.

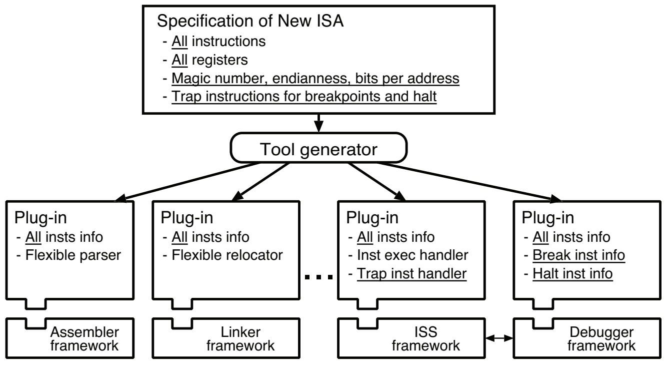

Fig. 4. Auto-retargeting of tools based on plug-in method.

method. The *foo* framework is the base part of the tool *foo* that handles machine independent tasks. The tool generator generates a plug-in for each tool from processor's specification which contains all instructions' information and deals with instruction specific processing by using the information. For example, the assembler framework takes care of file I/O and management of data structure, which calls the plug-in to parse and translate each mnemonic instruction to a binary code.

The tool frameworks are prepared by 1) choosing a GNU toolchain of an arbitrary ISA, 2) making sockets for a plug-in in each tool, and 3) deleting ISA dependent part from the tools.

Note that retargeting of GDB to new processors needs implementation of breakpoints. This was not an issue in the case of ISA extension in the previous section, for all the functionalities for dealing with breakpoints has been already defined in the GDB for the base processor. However, since handling of breakpoints is ISA dependent, it must be incorporated into the plug-ins in the case of toolchain retargeting. Thus the tool generation scheme in [14] must be extended so that it can deal with breakpoints in GDB.

B. *Implementation of Breakpoints of GDB*

The ISS implemented in GDB simulates the behavior of the processor by repeating the following steps.

1. Fetch an instruction.
2. Test if it is a trap instruction.

   If true, break execution and give control to GDB; otherwise, continue.
3. Execute the instruction.

In the ISA extension scheme in [14], the step 2 was assumed to be already implemented in the ISS for the base processor, while it must be generated into the plug-in for the ISS in the case of toolchain retargeting (since which instructions cause trap is ISA depemdent). For this purpose, ISA specification is extended to include the rules to identify the trap instructions to break or to halt the ISS, so that the tool generator can generate a function for the step 2 into the plug-in for the ISS.

When the trap is detected and the control is passed from the ISS, GDB handles breakpoints in the following way.

1. Saves the instruction at the breakpoint address, and puts a trap instruction at the address.
2. Stops the ISS at the trap instruction, and restores the saved instruction to the address.
3. On resume, makes the ISS to execute a single instruction, and again saves the instruction at the breakpoint and put a trap instruction at the address.

Since all the above procedure is machine independent, the breakpoint handler is retargeted by modifying the machine dependent part in the following way.

1. Let GDB recognize the trap instruction:

   Identify the binary code of the trap instruction for break, and store it into the data structure of GDB using the plug-in, so that GDB can find the code to write into the breakpoint address.

2. Stop the ISS on the trap instruction:

   Generate a function to test if a given instruction is a trap. Let the plug-in calls this function and if it returns true, then invoke the GDB function to handle the trap.

C. *ISA Specification*

The information of the instructions and registers are specified in the same way as in instruction set extension (Fig. 2 and Fig. 3). In addition, in the case of tool retargeting, the following items must also be specified:

- Magic number
- Endianness
- Number of bits per address
- Trap instructions to halt the ISS
- Trap instructions for breakpoints

D. *Example*

Fig. 5 shows an example of the ISA specification of the 16 bit processor MeDIX-I [15]. The primary features of the ISA, such as endianness, the address length, and the magic number, are specified as attributes of the `<name>` tag (lines 2–5). Registers (lines 7–12) and instructions (lines 15–31) are declared in the same format as in Fig. 2, except that the `<insn>` tag has the `iss_halt_insn` and `iss_break_insn` attributes, which indicates if the instruction is a trap instruction to halt the ISS and to stop the ISS at breakpoints, respectively.

V. Implementation and Experiments

A. *Implementation*

We implemented a tool retargeting system based on the proposed method by extending the system developed in [14]. The system is written in C++ and runs on Unix environments. From given ISA specification, it generates the plug-ins for the assembler, the disassembler, the linker of GNU Binutils, and the ISS and the debugger of GNU GDB. It supports both endianness, and both fixed and variable length instructions up to eight byte long. Generated tools can handle relocation of immediate address operands and both relative and absolute jumps. The generated GDB debugger supports breakpoints and tracing of instruction execution.

```
 1:  <Processor>
 2:  <name
 3:     endian="big"
 4:     bits_per_address="16"
 5:     magic_number="0x1010">MeDIX 1</name>
 6:  <nickname>medix1</nickname>
 7:  <register_type length="16">
 8:     GENERAL_type</register_type>
 9:  <register_bank type="GENERAL_type" size="16"
10:     prefix="r">GPR</register_bank>
11:  <register_bank type="GENERAL_type" size="1"
12:     prefix="pc">PC</register_bank>
13:  <insn_length>16</insn_length>
14:
15:  <insn iss_halt_insn="false" iss_break_insn="false">
16:     <mnemonic>ADD</mnemonic>
17:     <syntax>ADD %rd, %rs</syntax>
18:     <field type="opcode" length="4" value="0b1000">
19:        opcode0</field>
20:     <field type="opcode" length="4" value="0b0000">
21:        opcode1</field>
22:     <field type="GPR" length="4">rd</field>
23:     <field type="GPR" length="4">rs</field>
24:     <input>
25:        <operand type="GPR" width="16">rd</operand>
26:        <operand type="GPR" width="16">rs</operand>
27:     </input>
28:     <output>
29:        <operand type="GPR" width="16">rd</operand>
30:     </output>
31:  </insn>
32:  .....
33:  <behavior>medix1-isa.c</behavior>
34:  </Processor>
```

Fig. 5. Specification of a target processor.

B. *Experiments*

We took two different types of processors for experiments: the MeDIX-I processor [15], mentioned in Section D., and a 32 bit in-house RISC processor. From XML and behavior description of each processor, Binutils and GDB were successfully generated. Table I shows the summary of the experiments. MeDIX-I is a big endian processor whose data and instruction words are both 16 bit long (fixed). Its 48 instructions have a typical assembly syntax. In contrast, the 32 bit RISC deals with little endian data and the instruction word length varies from 16 to 48 bits. The instruction set consists of 461 instructions whose assembly syntax is in arithmetic form (i.e. `r2=r5+r7`). The XML and behavior description amounts to 818 and 660 lines, respectively, for MeDIX-I, and 10,018 and 8,568 lines for the 32 bit RISC. Our tool generator worked on different types of Linux distributions and also on different versions of Binutils and GDB, without modification on the plug-in generator, except for some minor adjustments for control scripts. Plug-in source code generation takes less than 1 second, which is negligible to the time for tool building.

C. *Limitations*

As compared with the full-fledged Binutils and GDB, those generated by our system have some limitations. First, all the functionalities in Binutils and GDB are not retargeted. We must implement plugs and plug-ins for the

TABLE I
Summary of experimental result.

	MeDIX-I	32 bit in-house RISC
Data/Endianness	16 bit/big endian	32 bit/little endian
Instruction	16 bits (fixed)	16/32/48 bits (mixed)
# of instructions	48	461
# of GPR	8	8
Assembly syntax	mnemonic	arithmetic
XML	815 line	10,018 line
Behavior	660 line	8,568 line
OS	Ubuntu 8.04	CentOS 4.8
Binutils	ver 2.16.1	ver 2.20
GDB	ver 6.4	ver 7.2
CPU (Memory)	Core2Duo 2GHz (1GB)	Xeon 1.8GHz (4GB)
Source generation	less than 1 sec	less than 1 sec
Build	340 sec	360 sec

necessary tools (which is not a difficult task). A major limitation is that the current XML specification does not capture procedure calling conventions so that generated GDBs are not capable of stack frame analysis and thus does not support C source level debugging. The linker does not support the relax option. VLIW processors are currently out of scope of our system.

VI. Conclusion

This paper has shown that the plug-in method [14] with enhancement for handling breakpoints is applicable to retargeting GNU Binutils and GNU GDB for newly developed processors from succinct architecture specification. Experimental results show that the method is flexible enough to handle various types of processors. It is our future challenge to enhance our method to handle C source code debugging.

Acknowledgment

Authors would like to express their thanks to Dr. Atsushi Atarashi of NEC, Dr. Takuji Hieda of Ritsumeikan University, and all the members of Imai Lab. of Osaka University and Ishiura Lab. of Kwansei Gakuin University, for their discussion.

References

[1] M. Imai, Y. Takeuchi, K. Sakanushi, and N. Ishiura: "Advantage and Possibility of Application-domain Specific Instruction-set Processor (ASIP)," *Trans. System LSI Design Methodology*, vol. 3, pp. 161–178 (Aug. 2010).

[2] H. Iwato, K. Sakanushi, Y. Takeuchi, M. Imai, A. Matsuzawa, and Y. Hirao: "A Low Power SoC for Pressure Measurement Capsules in Ambulatory Urodynamic Monitoring," in *Proc. Cool Chips XIII*, pp. 441–446 (Apr. 2010).

[3] Y. Takeuchi, K. Sakanushi, and M. Imai: "Generation of Application-domain Specific Instruction-set Processors," in *Proc. ISOCC 2010*, pp. 75–78 (Nov. 2010).

[4] R. E. Gonzalez: "Xtensa: A Configurable and Extensible Processor," in *Proc. IEEE Micro*, vol. 20, Issue 2, pp. 60–70 (Mar.–Apr. 2000).

[5] S. Kobayashi, Y. Takeuchi, A. Kitajima, and M. Imai: "Compiler Generation in PEAS-III: an ASIP Development System," in *Proc. Int'l Workshop on Software and Compilers for Embedded Processors* (Feb. 2001).

[6] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren: "A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models," in *Proc. DATE*, pp. 1276–1281 (Feb. 2004).

[7] CoSy, see http://www.ace.nl/.

[8] LISATek, see http://www.coware.com/.

[9] CGEN, the Cpu tools GENerator, see http://sources.redhat.com/cgen/.

[10] M. Abbaspour and J. Zhu: "Retargetable Binary Utilities," in *Proc. 39th DAC*, pp. 331–336 (June 2002).

[11] A. Baldassin, P. Centoducatte, and S. Rigo: "An Open-Source Binary Utility Generator," *ACM Trans. Design Automation of Electronic Systems*, vol. 13, no. 2, art. 27 (Apr. 2008).

[12] GNU Project, see http://www.gnu.org/.

[13] GPL, see http://www.gnu.org/licenses/gpl.html.

[14] T. Kumura, S. Taga, N. Ishiura, Y. Takeuchi, and M. Imai: "Software Development Tool Generation Method Suitable for Instruction Set Extension of Embedded Processors," *IPSJ Trans. System LSI Design Methodology*, vol. 3, pp. 207–221 (Aug. 2010).

[15] Y. Takeuchi, H. Ohsawa, K. Kondo, H. Iwato, K. Sakanushi, and M. Imai: "Low Energy MDPC Implementation using Special Instructions on Application Domain Specific Instruction-set Processor," in *Proc. APSIPA ASC 2010*, pp. 47–52 (Dec. 2010).

[16] Tensilica, Inc.: *Tensilica Instruction Extension (TIE) Language Reference Manual*, (Nov. 2006).