Retargetable Instruction Selection and Generation of GCC Back-end Using High-level Processor Models

Abstract

Automatic generation of compiler back-end has been studied for a while. However, recent growth in embedded systems has generated renewed interest in this area of technology. An embedded system is an application specific electronic system that typically contains one or more CPUs and some software running on the CPUs. Due to application specific nature of the system, designers often want to use a CPU with an instruction set that is optimized for the application. As a result, a large number of CPUs are available for the embedded systems, each having some features that make it particularly attractive for one application domain. Even general purpose embedded processors like ARM [15] support a co-processor interface, using which designers can extend the ARM instruction set with custom application specific co-processors. Vendors like Tensilica [14] and ARC [13] provide configurable CPU cores, whose instruction set can be customized by the designer to suit his/her application needs. However, to make use of the customized instruction set of a CPU, one needs software generation tools that can generate code for the custom instruction set. One approach of attacking this problem is to automatically generate software generation tools like compiler, assembler, linker, etc, from a simple model of the processor.

Sim-nML [1] is a high-level language for processor modeling. It is possible to generate assembler, disassembler, functional simulator, etc from Sim-nML processor models [2], [3], [4]. In this paper we discuss generation of a compiler from Sim-nML processor models.

A compiler is typically partitioned into two parts-- a front-end, which is specific to the source language but independent of the target instruction set and a back-end, or a code generator, which is source language neutral but is dependent on the target Instruction Set Architecture (ISA). To automatically generate a compiler for a target, one has to generate the back-end while the same front-end can be used for all the targets. The tasks that an automatic back-end generator performs are instruction selection, resource allocation and instruction scheduling. In the paper, we discuss the instruction selection techniques from Sim-nML processor models and their application in porting GNU Compiler Collection (gcc) [12] to a new target.

I. Introduction

We have witnessed a phenomenal growth in silicon capacity of Integrated Circuits (ICs) over past two decades and the trend still continues. This growth makes it possible to embed a complex electronic system into small appliances, such as a mobile phone, or a set-top box, etc. Such electronic systems are referred to as embedded systems, which typically consist of one or more CPUs, software to run on them, and other non-programmable electronic hardware. Unlike a general purpose computer, they are designed for a particular application and are as diverse as their possible applications. Some of the common applications where embedded systems can be found are mobile phones, digital cameras, music players, hand-held computers, set-top boxes, etc. Designers of the embedded systems try to leverage their application specific nature by optimizing the system for the target application. It is a common practice to select a CPU for an embedded system whose Instruction Set Architecture (ISA) is particularly suitable for the target
application. A very large number of CPUs, ranging from simple 8-bit microcontrollers to very powerful 64-bit RISC processors, are available for embedded system design. It is also common for embedded CPU vendors to provide support for some customization. As an example, ARM CPUs [15] support a co-processor interface, which can be used to extend ARM ISA by adding a custom co-processor. Configurable CPUs, whose instruction sets can be tailor made for an application, are becoming popular too. CPU vendors like Tensilica [14] and ARC [13] are providing this technology to the embedded systems designers.

One of the key challenges of using an application specific ISA for an embedded system is software development. To support software development for a target ISA one needs compiler, assembler, linker, debugger, etc for the target. It is overkill for the embedded system designer to develop complete software development tools for his/her optimized ISA. One approach for solving this problem is to automatically generate the software development tools from a simple model of the target ISA. Sim-nML [1] processor modeling language, developed in IIT, Kanpur, can be used to write processor models, from which tools like assembler, disassembler, functional simulator, [2], [3], [4] etc can be generated. In this work, we have developed techniques for generating a compiler from Sim-nML processor models.

A compiler is typically partitioned into two parts. The front-end is responsible for lexical analysis; parsing and target independent optimizations of the source program and is specific to the source language. However, it is target independent. The front-end converts the source to an intermediate form, which is matched against the target instruction patterns by the back-end or the code generator. The back-end also performs register allocation, instruction scheduling and additional target dependent optimizations and generates the target program. Thus the back-end is source language neutral but target dependent. To port a compiler to a new target, it is necessary to port the back-end of it.

To automatically generate a compiler for a target ISA, one has to generate only the back-end or the code generator. Such a tool reads a model of the target ISA and extracts instruction patterns that can be matched against the intermediate form of the source program. This is known as instruction selection. In addition, it has to generate information about registers and other resources like number and availability of integer ALUs, Floating Point Units (FPUs), instruction latency, etc to assist the back-end in register allocation, instruction scheduling and target dependent optimizations.

In this paper we discuss the instruction selection techniques from Sim-nML processor models. In Sim-nML language, one describes semantic of an instruction using high-level C-like constructs, which is used for functional simulation of the processor. So our key challenge was to extract simple patterns, which can be matched against instruction templates, from complex models of instructions. The tool we have developed simplifies the instruction models and outputs the extracted instruction patterns in the form of a gcc (GNU Compiler Collection) md file [12]. The md file is used by the gcc back-end to match the target instruction patterns with gcc's intermediate form. The tool also generates the information about the target registers in the form of target description macros. Additionally, we have put some manual efforts and build a cross gcc for the target.

In the next section we give a brief description of Sim-nML processor modeling language. In section 3 we discuss instruction selection from Sim-nML. Our experience of porting gcc using the generated md file is described in section 4. In section 5 we present the results and conclude this paper.

Sim-nML technology has earlier been described in [1], [2], [3], [4], [5], [6]. The approach of automatically generating compiler from a description of a processor has been reported in [7], [8], [9], [10], and [11].
II. Sim-nML Processor Modeling Language

Sim-nML [1] is a language for describing the ISA of a processor. A Sim-nML model of a processor has a tree-like structure. A traversal of the tree from the root to a leaf gives definition of an instruction. The relationship between to nodes in the tree is expressed as rules. There are two kinds of rules-- *and rule* and *or rule*. In an *and rule*, the attributes or properties of the parent node are defined in terms of those of the children. An *or rule* is a collection of *and rules*, each having exactly one child and the same parent. Attributes of an instruction that are defined by a rule are assembly syntax, binary image, semantic action, etc. The rules that constitute an addressing mode or a part of it are called *mode rules*. Other rules are called *op rules*. Mode rules carry an expression that provides the value of the operand.

In Sim-nML one can also describe registers, memory, and temporary storage elements needed for writing the action of an instruction. Actions are described in a C like language.

Figure 1 shows an example of a simple Sim-nML model. It describes a processor with two instructions-- add and swap, 32 general purpose 32-bit registers, and a byte addressable memory with 32-bit address. The keywords *reg*, *mem*, and *var* are used, respectively, to model registers, memory, and temporary variables. The *mode* keyword defines a mode rule that models register addressing mode. Likewise, *op* keyword is used to define op rules. op rule `instruction' is the root of the tree. op rule `add' describes attributes of an add instruction. The attributes *syntax, image* and *action* are defined to mean assembly syntax, binary image and semantic action of an instruction. It has two registers as its arguments. It adds contents of them and stores in the 1st register. Similarly, op rule `swap' models a swap instruction. It shows one example use of temporary variables.

More detailed information about of Sim-nML language can be found in [1], [3], and [6].

```plaintext
//Types
type unsignedByte = card(8)
type unsignedWord = card(32)

//Registers GPR[0] to GPR[31]
reg GPR[31, unsignedWord]

//Memory
mem M[2**32, unsignedByte]

//temporary variable
var temp[1, unsignedWord]

//Addressing mode:
//this is a mode and-rule
mode register(ind:card(5))=GPR[ind]
syntax = format("R%d", ind)
image = format("%5b", ind)

//instructions
//this is an op or-rule
op instruction = add | swap

//this is an op and-rule
op add (r1 : register, r2 : register)
syntax = format("add %s, %s",
   r1.syntax, r2.syntax)
image = format("000101%s%s",
   r2.image, r1.image)
action = {
   r1 = r1 + r2;
}

//this too is an op and-rule
op swap(r1 : register, r2 : register)
syntax = format("swap %s, %s",
   r1.syntax, r2.syntax)
image = format("000110%s%s",
   r2.image, r1.image)
action = {
   temp = r1;
   r1 = r2;
   r2 = temp;
}
```

Fig. 1: A simple Sim-nML model

III. Instruction Selection from Sim-nML Processor Models
A Sim-nML model of a processor describes the instructions of the processor in a tree-like structure. The tree is flattened [5] to obtain definition of each instruction. Instruction selection involves matching flattened action of each instruction with the predefined instruction patterns. Matched instructions can be used by a compiler for code generation. However, in general, the action definitions are fairly complex and can not be matched as is with any existing instruction pattern.

For example, consider the example of the instruction shown in Figure 2. The op rule `branch_or_call' models both the conditional branch and call instructions. Both branch and a call instructions check if 'flagRegister' is set and if set, assigns the target address to the program counter register, GPR[15]. A call instruction additionally saves the address of the next instruction to link register, GPR[14]. The target address is computed as a sum of a base register and an immediate offset. Note that a the link register can also be a base register and so, it is necessary to compute the target and save it in a temporary variable, 'target_if_taken', before the link register is modified.

```
op branch_or_call(link : card(1), b : registerIndex, o : immediateOffset)
action = {
    //GPR[15] is program counter.
    //every instruction increments GPR[15] by 4
    //target address if branch is taken.
    //target_if_taken is a temporary variable
    target_if_taken = GPR[b] + o;
    //save address of the next instruction
    //to link register, if it is a call instruction
    if (link == 1) then
        GPR[14] = GPR[15];
    endif;
    //evaluate the condition
    //branch is taken if the flagRegister is 1
    //actual_target is a temporary
    if (flagRegister == 1) then
        actual_target = target_if_taken;
    else
        actual_target = GPR[15]
    endif;
    GPR[15] = actual_target;
}
```

**Fig. 2:** action of a branch or call instruction

Note that it is difficult to match the action of this instruction with any predefined pattern. It is the presence of branches and temporaries that makes matching a difficult task. So we need to apply simplifying transformations to remove branches and temporaries before actions can be matched against the patterns.

**III.1 Eliminating Temporary Variables**

A use of a temporary variable can be eliminated by replacing it with its most recent definition. However, definition of a temporary may be killed if any one of the variables or state elements, like registers or a
memory words, appearing on the RHS of the definition is redefined. For example in Figure 2, temporary variable `target_if_taken' is a sum of a base register and an immediate offset. This definition is potentially killed by the statement that assigns to the link register, GPR[14], because the link register may be the base register as well. In this scenario, replacing the use of `target_if_taken' with its definition will give incorrect result. In fact, it may never be possible to remove a temporary in sequential model of control flow. An example of this is found in the definition of the swap instruction in Figure 1. So to eliminate temporaries we transform a sequential block of action statements to a block of parallel statements where all the right-hand side expressions are evaluated before any value is assigned to any state element or variable\[5\], \[6\]. Then we safely propagate most recent definitions of all variables and state elements to their uses. Figure 3 shows the parallelized model of the swap instruction.

```
action = {
    r1_out = r2_in;
    r2_out = r1_in;
}
```

Fig. 3: Parallelized swap instruction

Sometimes, it is not possible to find the most recent definition of a temporary or a state-element because of presence of branches. For example, in Figure 2, the definition of the temporary `actual_target' depends on whether the register `flagRegister' is set. If set, `actual_target' is `target_if_taken'. Otherwise, it is the address of the next instruction. To avoid the problem, we move the code that follows a branch into both the paths of the branch and propagate the definition to uses \[6\]. So the 2\textsuperscript{nd} branch in Figure 2, after this transformation, becomes as shown in Figure 4.

```
if (flagRegister == 1) then
    GPR[15] = target_if_taken;
else
endif;
```

Fig. 4: Temporary elimination after code motion

III.2 Eliminating Branches

A branch can be eliminated if outcome of the branch is known statically, i.e., its outcome doesn't depend on a state element. It is common to model multiple similar instructions as a single instruction so that code can be reused. For example, the op rule `branch_or_call' models both a conditional branch and call instructions. Such instructions have a branch whose outcome depends on the value of a basic type parameter (a parameter which is not another op or mode rule). When a code generator emits this instruction, it assigns values to all the basic type parameters and so the outcome of the branch is known at code generation time. Such an instruction can be splitted into two different instructions, each with one path of the branch \[6\]. So `branch_or_call' instruction can be splitted into a branch instruction and a call instruction. For the branch instruction, the parameter `link' is always 0, while for the call instruction, the parameter is always 1.

Branches can further be eliminated by performing constant folding-- evaluating constant expressions. Such a transformation leads to branches which check the value of a constant and so, are known statically \[6\]. We eliminate these branches by eliminating the paths that are never taken.

III.3 Eliminating PC Update

Every instruction routinely updates program counter. Such statements do not describe the functionality of the instruction and so are removed from the action sequence \[6\].

Figures 5 and 6 show the branch and call instructions which are obtained from the
'branch_or_call' op rule of Figure 2 by applying the simplifying transformations.

```
action = {
    if (flagRegister == 1) then
    endif;
}
```

Fig. 5: Branch instruction obtained after simplification

```
action = {
    GPR[14] = GPR[15];
    if (flagRegister == 1) then
    endif;
}
```

Fig. 6: Call instruction obtained after simplification

III.4 Matching Instructions

Once the instructions are simplified, we try to match them with against a set of predefined instruction patterns. For example, Figures 7 and 8 show the patterns for an add instruction and a branch if set instruction respectively.

```
Operand3 = operand2 + operand1
```

Fig. 7: Pattern for 'add' instruction

```
if flag == 1 then
    pc = operand1 + operand2
endif
```

Fig. 8: Pattern for a 'branch if set' instruction

The pattern for 'branch if set' instruction readily matches that of the branch instruction shown in Figure 5.

IV. Porting GCC to a New Target

The GNU Compiler Collection or gcc [12] supports a number of source languages, like C, C++, ADA, Java, etc, as well as a wide variety of target processors, like Intel x86, ARM, UltraSparc, PowerPC, MIPS, etc. gcc has a front-end, which is specific to the source language but independent of the target architecture, and a back-end, which is independent of the source language but specific to the target architecture. To port gcc to a new target, one has to write a machine description in the form of an md file and a couple of C source and header files.

IV.1 GCC Machine Description

The md file in gcc machine description controls the generation of gcc's intermediate representation of a source program, called RTL representation, as well as generation of the target assembly instruction by gcc's back-end. The define_expand keyword in md file tells gcc what RTL pattern to emit to perform a particular operation. For example, Figure 9 shows a define_expand pattern, that is to be used by gcc to perform addsi3 or 3-operand add of single integers.

```
(define_expand "addsi3"
(set
  (match_operand:SI 0 "general_operand" "")
  (plus:SI
    (match_operand:SI 1 "general_operand" "")
    (match_operand:SI 2 "general_operand" "")
  )
)
```

Fig. 9. Pattern for addsi3

In addition, the keyword define_insn is used to tell gcc back-end what assembly instruction to emit for a matching RTL pattern. So for every RTL pattern that may be generated by define_expand, there should be a matching define_insn pattern. Figure 10 shows a define_insn pattern, that matches
the pattern for addsi3 shown in Figure 9.

In addition to the md file, gcc machine description also has a few C source and header files which describe register classes, ranges of immediate operands, supported addressing modes, ABI etc.

IV.2 Generation of GCC Machine Description

We have developed a tool called genmd2 which reads a Sim-nML processor model and a configuration file and generates a gcc machine description. The configuration file provides information about the program counter, link register, condition code register, etc.

After reading the Sim-nML model and configuration information, genmd2 flattens the Sim-nML model to obtain definition of each instruction of the processor. It then performs various simplifications on the instruction actions and matches the simplified instructions against the predefined instruction patterns. Finally, it generates a gcc machine description for the processor using the matched instructions.

Every predefined pattern that matches at least one instruction is generated in the md file using define_expand. For every instruction that matches such a pattern, genmd2 generates a define_insn.

```
(define_insn
  (set
    (match_operand:SI 0 "register_operand" "a")
    (plus:SI
      (match_operand:SI 1 "register_operand" "a")
      (match_operand:SI 2 "register_operand" "a")
    )
  )
  "add %1, %2, %0"
)
```

Fig. 10. Assembly instruction that can be used for addsi3

genmd2 also generates some target description macros from the register and addressing mode information present in the Sim-nML model.

V. Results

We have used genmd2 to generate a gcc machine description of 64 bit Sparc V9 ISA. The generated md file had instructions for arithmetic-logic operations (e.g. add, sub, div, udiv, and, xor), data movement (mov), control transfer (beq, bnc, bgt, ble), comparison (tst), etc. However, generated md file was not complete as some of the more complex instructions were not matched. So we hand edited the md file to add some additional patterns. A few C files were hand coded as they had information about ABI, calling conventions, etc, which are not present in the processor model. A few header files specific to the target operating system (Solaris 2) were reused from an existing port of gcc. Figure 11 shows the lines of code generated by genmd2 as well as added manually to complete the gcc port.

<table>
<thead>
<tr>
<th>File</th>
<th>Obtained from</th>
<th>Lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>target.md</td>
<td>genmd2</td>
<td>1922</td>
</tr>
<tr>
<td>target.md</td>
<td>after manual editing</td>
<td>2694</td>
</tr>
<tr>
<td>target.h</td>
<td>genmd2</td>
<td>261</td>
</tr>
<tr>
<td>target.c</td>
<td>genmd2</td>
<td>260</td>
</tr>
<tr>
<td>sparc.h</td>
<td>hand-coded</td>
<td>3617</td>
</tr>
<tr>
<td>sparc.c</td>
<td>hand-coded</td>
<td>1489</td>
</tr>
<tr>
<td>sol2.h</td>
<td>reused</td>
<td>185</td>
</tr>
<tr>
<td>sysv4.h</td>
<td>reused</td>
<td>221</td>
</tr>
<tr>
<td>svr4.h</td>
<td>reused</td>
<td>980</td>
</tr>
</tbody>
</table>

Fig. 11. Lines of code in the gcc port of Sparc v9

With this machine description, we built a gcc for Sparc v9 and compiled a few simple C programs using it.
VI. Conclusion

In this paper we have presented a technique for instruction selection from Sim-nML processor models and its application in generating gcc machine description. The generated machine description, however, was not complete and required some manual editing before a gcc port could be built. We feel that gcc doesn't provide an ideal infrastructure for compiler generation as it is not designed for this purpose. However, we have taken advantage of gcc's existing code-base to prove our instruction selection techniques from Sim-nML and the fact that it is possible to generate a compiler for a processor from the Sim-nML model of it.

This work can be further extended by automating resource allocation, instruction scheduling and target dependent optimizations from Sim-nML processor models. Instruction selection can also have other applications like analysis of an ISA for completeness or suitability for a particular embedded application, generation of translators that convert one machine language program to another, generation of HDL from a binary program, etc.

References