INSTRUCTION SET GENERATION CONSIDERING THE INTERMEDIATE REPRESENTATION OF COMPILERS

Jong-Yeol Lee, In-Cheol Park
KAIST, Dept of EE
373-1 Gusong-dong, Yusung-gu, Daejon
Tel : 042-869-4405 Fax : 042-869-4410
Email : jylee@ics.kaist.ac.kr, icpar@ics.kaist.ac.kr

Abstract

This paper presents a new method to define an instruction set by considering the intermediate representation of compilers. In the proposed method, instructions are defined to fully exploit the intermediate representation of compilers to achieve object codes with small size and high performance. To verify the proposed method, the intermediate representation of GCC is analyzed and new instructions targeting for small code size and cycle count are defined based on the analysis. Experimental results on benchmark programs show that the proposed method is effective in reducing code size and improving performance.

1. Introduction

As the size of software programs is becoming larger, programming in assembly languages is being diminished and most software programs are now designed and implemented in high-level languages. Even in embedded systems, there is a growing tendency to use high-level languages. Hence, the efficiency of compilers that support high-level languages is one of the most important factors that determine the performance of processors.

Since the instruction set of a processor affects the quality of compiled codes and the complexity of building a compiler, understanding compiler technology is important to design and implement an efficient instruction set. However, designers usually pay little attention to compilers when defining the instruction set of a new processor, and compilers are often designed after the instruction set is completely defined. If the instruction set is defined without considering compiler technology, it may lead to poor compilation performance, i.e. large object code size and cycle count. This is significant in embedded applications that need real-time performance and small object code size. In the application, the code size is directly related to the size of memory.

Traditionally, instruction sets have been defined based on designers' intuition, the analysis of high-level languages, or architectural measurements. In [1] and [2], the authors suggested software-oriented architectures by studying the properties of high-level languages. The authors of [1] proposed a 32-bit microprocessor architecture where most Forth primitive operations are executed in one cycle. In [2], an instruction set was designed to support basic functional operations and data types used in C language. Attempts to incorporate high-level language features in an instruction set lead to complex instructions. Historically, the approach was not successful because such complex instructions make the underlining hardware architecture complex and make it difficult to use high performance schemes such as pipelining. A quantitative approach for an instruction set design is provided in [3], which describes some instruction measurements on memory addressing, operations, type and size of operands, and encoding of instructions and explains how to design an instruction set based on the measurements. Fig. 1a and Fig. 1b show the traditional approaches.

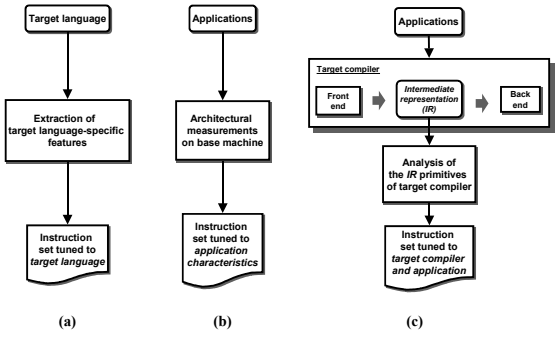


Fig. 1. Approaches to define instructions. (a) Approach in [1] and [2]. (b) Approach in [3]. (c) Proposed approach.

In contrast to the previous approaches, we propose a new approach where an instruction set is designed with considering the target compiler to achieve high performance and small code size. In the proposed approach, instructions are defined by analyzing the intermediate representation (IR) of the target compiler. The IR is a kind of abstract machine language that can express the target-machine operations without committing to a full of machine-specific details. It is also

independent of the details of the high-level languages. After analyzing the IR codes for a set of benchmarks, instructions are defined to implement the behaviors of frequent primitives in IR codes. The instructions defined in this way can save the burden of building a new compiler and furthermore make it possible for the target compiler to produce efficient codes. The proposed approach is different from the approaches of [1], [2], and [3] in that instructions are defined based on the analysis performed on the IR. Since the IR represents simple operations, the instructions obtained from the analysis of the IR code are not complex but simple enough to be required as reduced instruction set computer (RISC) instructions. This is a clear difference from the previous approaches that have generated complex instructions from the analysis of high-level languages. Fig. 1c shows the proposed approach.

To verify the proposed method, we defined a new instruction set by analyzing the IR of GNU C compiler (GCC) which is one of the most popular portable compilers ported for almost all commercial processors and many in-house embedded processors. The code quality of GCC is reported to be better than commercial compilers for some machines. The IR of GCC is a register transfer language (RTL) that has a LISP-like form.

The rest of this paper is organized as follows. We describe motivations for this work in section 2. The proposed method is outlined in section 3. The analysis of the IR primitives of GCC generated when compiling some benchmarks is explained in section 4 and new instructions defined based on the analysis are described in section 5. The experimental results are presented in section 6 to show the effectiveness of the proposed method. Finally we provide conclusions in section 7.

2. Motivation

Compilers often fail to generate efficient codes in terms of code size and cycle count on processors whose instructions are designed without considering compilers. In a complex instruction set computer (CISC) machine, instructions are so complex that only a subset of simple instructions appears in the compiler-generated codes. On the other hand, RISC machines are proposed to get higher performance of compilers by providing simple instructions that compilers can easily exploit. However, the compilers generate large-sized codes for RISC machines than for CISC machines due to the simplicity of instructions. This is, however, not a major problem in compilers for many RISC machines and compiler designers made little efforts to provide optimizations to reduce code sizes.

This inefficiency of compilers can be explained by looking into the operation of compilers. On a processor with instructions defined in the traditional approaches without considering the IR, a compiler may show poor performance when there are mismatches between the IR primitives of the compiler and target machine instructions. As the IR is an abstract machine language of a generic machine and there is not always a one-to-one mapping between IR primitives and target machine instructions, one IR primitive may be translated into more than one target machine instructions and vice versa. This may lead to inefficient codes. For example, an operation in the source code that can be implemented by one machine instruction is translated into a sequence of IR primitives in front-end if there is no IR primitive corresponding to the complex machine instruction. The IR primitives of the sequence are translated into corresponding simple machine instructions in the final code generation step and hence the complex instruction cannot be generated. On the other hand, when an IR primitive is not implemented by a machine instruction, a sequence of machine instructions is needed to implement the IR primitive, resulting in large object code size and cycle count. To avoid the inefficiency, machine instructions should have the same complexity as IR primitives.

The instructions defined by the analysis of IR codes can be readily used in code generation and can eliminate some machine dependent optimizations, as the IR primitives can be translated into the matched target machine instructions. Machine dependent optimizations such as instruction combining [5] and peephole optimization [7] may be skipped since there is no discrepancy between IR primitives and target machine instructions. The peephole optimization replaces a sequence of instructions by a shorter or faster sequence. This can be effective when a processor has complex instructions that can be implemented by a sequence of simpler instructions. A compiler first generates code with only instructions that can be represented by IR primitives and later checks if a complex instruction can replace a sequence of instructions. However, when instructions are defined based on the IR, the IR primitives can represent all the instructions and the peephole optimization can be skipped.

3. The Proposed Method

In the proposed method, an instruction set is defined as follows. First, IR codes are generated by compiling a set of benchmarks. The IR codes are analyzed statically and dynamically to produce various data needed for the definition of a new instruction set. In static analysis, the generated IR codes are examined to find the number of appearance of each IR primitive in the IR codes. The operands of the primitives are divided into several groups and the frequency of the operand groups and addressing modes of the memory operands is also collected. In dynamic analysis, a compiler is modified to generate executables that include a counter for each IR primitive and the generated executables are executed to report the dynamic frequency of the IR primitives, operand groups, and addressing modes. The analysis results also include the size of displacements of memory operands and the size of branch displacements and call addresses. Lastly, the new instructions are defined based on the static and dynamic analysis of the IR primitives. Machine specific features such as instruction encoding and operation parallelism are taken into account in defining the instructions.

4. Analysis of GNU IR Primitives

In this section, we analyze the IR primitives of GNU C compiler and describe how to improve SPARC instructions. Fig. 2 shows a simplified structure of GCC. Given a source code, GCC performs syntax analysis, semantic analysis, and parsing to construct a parse tree. The parse tree is translated into an IR code called *RTL code* at the preliminary code generation step. Register allocation, instruction scheduling, and various optimizations such as common sub-expression elimination, jump optimization, are performed on the generated RTL code. Finally, the optimized RTL code is translated into a target machine code.

Since GCC is a kind of retargettable compiler, re-writing Machine Description (MD) can modify the back end of the compiler. The MD contains a processor description consisting of an instruction pattern description and a functional units description written in RTL. Some of the instruction patterns have names that are meaningful only in the preliminary code generation step. These are called *standard patterns*. The standard patterns consist of move patterns, arithmetic patterns, sign or zero extension patterns, call-related patterns, bit-manipulation patterns, branch and compare patterns, and type casting patterns. Giving one of standard names to an instruction pattern tells how the RTL generation pass can use the pattern to accomplish a certain task. In the preliminary code generation step, only the standard patterns are considered to generate a RTL code. Hence, the standard patterns are primitives of RLT codes. For example, an integer addition in the source code is translated to a standard pattern named as "addsi3". The name "addsi3" informs the preliminary code generator of the addition of two integers. At the final step of compilation the "addsi3" pattern is translated into a target machine instruction that performs integer addition.

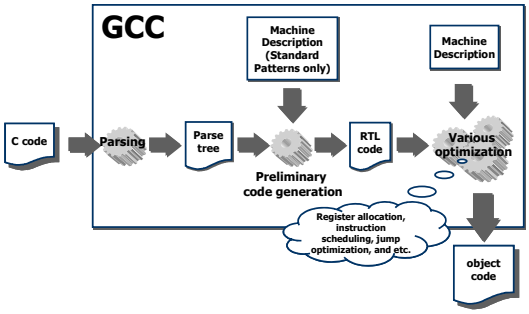


Fig. 2. Structure of GNU C compiler.

In order to measure the static and dynamic frequencies of standard patterns, we modified GCC to insert a counter for each standard pattern used into executables. We compiled MediaBench and SPECint95 benchmarks by using the modified GCC. The generated executables are executed using Shade analyzer [6], which was modified to report the dynamic frequency of each standard pattern by accumulating the counter values.

4.1. Static Analysis

Static analysis shows that four arithmetic and logical standard patterns, compare, add, and shift patterns, cover 17.1% of total patterns. The patterns except the four patterns cover less that 1% of total patterns. SPARC has corresponding instructions for all the arithmetic, logical, and control flow standard patterns. Hence, defining new instructions for those patterns is not needed.

Fig. 3a shows the ratio of the number of each standard pattern over that of total patterns. The move patterns take the largest portion covering almost 60% of total patterns. This is a typical result commonly found in RISC machines such as SPARC. Since the most frequent are move patterns, they should be treated in more

detail. The addressing modes of memory operands and the sizes of displacements are investigated. The addressing modes supported in the standard patterns are shown in Fig. 3b. The addressing modes with displacements are found in about 25% of total patterns. We measure the sizes of constants in the move patterns that move constants into registers so as to define new move instructions that efficiently moves constant data to their destinations. It is found that 1.5% of the total patterns is related to operand extension that performs sign-extension or zero-extension. The analysis also shows that the distribution of the bit width of the displacements of memory addresses should be larger than 13 bits. In SPARC instruction set, the size of the constant field is 13 bits. Hence, a constant larger than 13 bits should be loaded by using two instructions. To increase the range of constants that can be loaded by one instruction, we define a new instruction that can load 22-bit constants. The instruction encoding restricts the size of the constant field of the new instruction.

Another important characteristic of compiler-generated codes is that a small number of identical instruction pattern sequences appear repeatedly. To use this characteristic in reducing the code size, we investigate the occurrence of the pattern sequences and the result is shown in Fig. 3d.

4.2. Dynamic Analysis

Dynamic analysis shows that 13-bit displacement of a branch suffices and the address field should be larger than 20 bits. In SPARC, since the size of the branch displacement and the address field in a call instruction is 22 bits, no new instruction is needed for the call and branch patterns.

As in the static analysis shown in Fig. 3a, mov pattern takes the largest portion covering about 50% of total patterns. The dynamic distribution of the operands of mov patterns is shown in Fig. 4b. In this case, the most frequent operand type is (reg-reg) whereas in static analysis, the most frequent type is (reg-mem). This is a result of compiler optimizations that try to maintain operand values in registers as much as possible in the portions of the code estimated to be executed frequently.

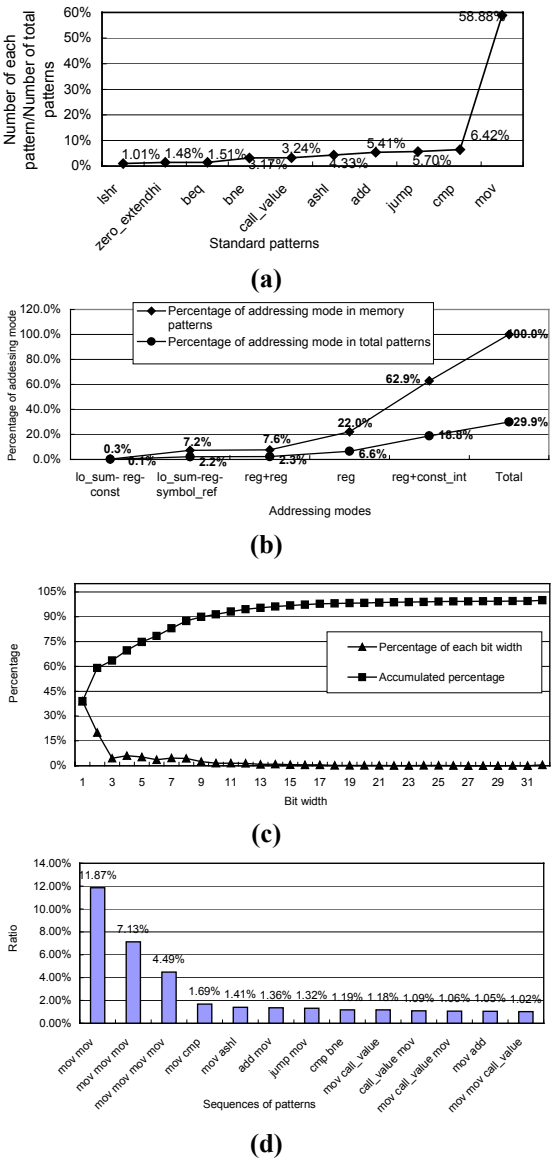
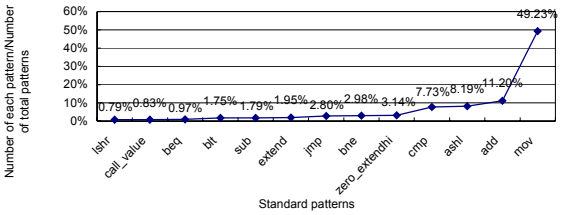


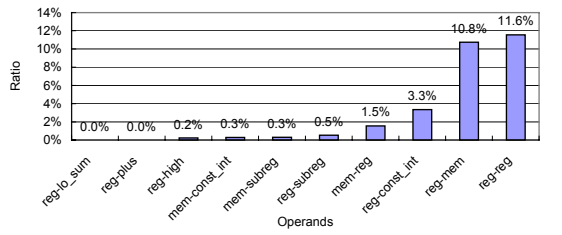
Fig. 3. Static analysis of standard patterns. (a) Distribution of standard patterns of GNU C Compiler. (b) Percentage of each addressing mode in total patterns and in memory patterns. (c) Bit width of constants in move patterns that move a constant into a register. (d) Consecutive patterns appearing two or more times.

Fig. 4c shows the addressing modes of memory patterns. The addressing modes with displacement are about 76% of memory patterns and 9.8% of total patterns. Fig. 4d shows the dynamic counts of the bit width of constants in reg-const_int patterns. The static percentage of constants larger than 13 bits is 4.6%. However, in Fig. 4d, the dynamic percentage of such constants is 26% of total constants appearing in reg-const_int pattern. This means that the new

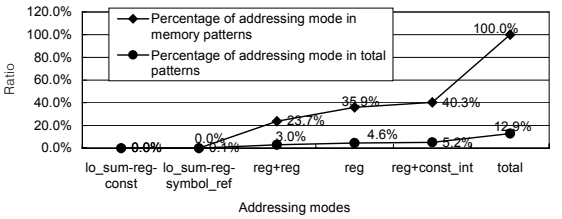
instruction with an extended constant field whose size is 22 bits is effective in reducing cycle count.



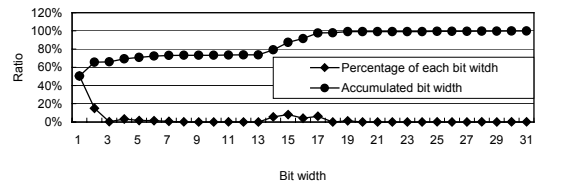
(a)



(b)



(c)



(d)

Fig. 4. Dynamic analysis of standard patterns. (a) Dynamic distribution of standard patterns of GNU C Compiler. (b) Dynamic frequency of operands in the mov pattern. (c) Dynamic percentage of each addressing mode in total patterns and in memory patterns. (d) Bit width of constants in move patterns that move a constant into a register.

5. New Instructions

In this section, new instructions defined based on the result in section 4 are presented. Considering the 32-bit instruction encoding of SPARC, the instructions are defined to reduce the code size and cycle count on SPARC.

The new instructions are an absolute instruction, three sign-extension instructions, three zero-extension instructions, data movement instructions, instructions for pattern sequences and a instruction conditionally set the destination. The absolute instruction is defined to exploit a standard pattern that calculates the absolute value of an operand. In SPARC, no integer mode ABS instruction is provided.

The scc standard pattern stores zero or non-zero value in its destination according to the condition specified by "cc". SPARC has no single instruction that can be used to implement these standard patterns. Hence, these standard patterns are implemented using branch and compare, or arithmetic instruction with condition codes. We define new instructions with this behavior.

The extension instructions are defined to implement sign-extension and zero-extension operations represented by the standard patterns "extend*mn*" and "zero_extend*mn*". Fig. 4a shows that the dynamic execution counts of extend and zero_extend patterns cover 5% of total execution count. Without these instructions, when the both operands are in registers, logical instructions or shift instructions are used.

Three kinds of move instructions are defined. The first move instruction is to move a sub-word of a register source into a destination. The other part of the destination must remain unmodified. The second move instruction moves a constant to a register. Although it is evident in the observation that the size of constant should be as large as possible, the encoding restricts the size of constant to 22 bits. The last move instruction is to move a constant to a memory location. It is shown in the analysis that 88% of the value of the constant is zero in a pattern that moves a constant into memory. Therefore, the move instruction is defined with eliminating the constant field and enlarging the displacement field.

Some instructions are defined to represent a sequence of patterns to reduce code size.

6. Experimental Results

The proposed method was implemented on an Ultra-SPARC workstation. We first modified GCC and compiled MediaBench and SPECint95 using the modified GCC to collect the data presented in section 4. After collecting data, we defined new instructions presented in section 5. We made a model of a virtual processor whose instruction set includes the new instructions as well as SPARC instructions. We constructed an assembler and a linker for the virtual processor to compile and link multiple C files.

6.1. Code Size Reduction

To measure the code size reduction with the new instructions, benchmarks were compiled with SPARC GCC. The code size resulted from the SPARC GCC was compared to the code size generated from the GCC ported to the virtual processor. Due to the newly defined instructions in the virtual processor, the code size was reduced as shown in Fig. 5. The maximum code size reduction is 15.9% with li that is a Lisp interpreter written in C language. The average code size reduction is about 8.5%.

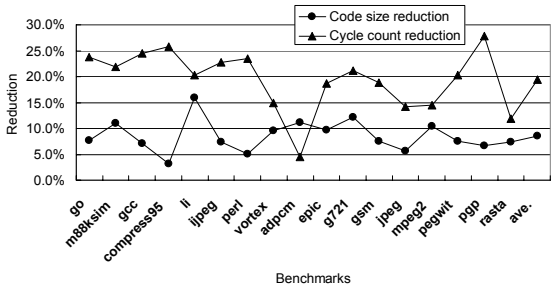


Fig. 5. Code size reduction and cycle count reduction with all new instructions. The first eight benchmarks are from SPECint95 and the last nine benchmarks are from MediaBench.

6.2. Cycle Count Reduction

We estimated the cycle count reduction by simulating the benchmarks on a SPARC instruction set simulator and adding up the execution counts of instructions. We used Shade analyzer [6] for instruction-level simulation and the analyzer reported the execution counts of instructions. The benchmarks were compiled with SPARC GCC and the GCC ported to the virtual processor. The generated executables were simulated with Shade analyzer to report the execution count of each instruction. The results are shown in Fig. 5, where the average reduction is about 20% and the maximum reduction is about 27%. In Fig. 5, we can recognize that the new instructions are significantly effective in reducing the cycle count.

7. Conclusion

In this paper, we presented a new approach to define instructions by considering the intermediate representation of compilers. By generating and investigating the intermediate representation for a set of benchmarks, we can find the static and dynamic occurrences of the IR primitives. Based on the analysis of intermediate representation, we can define new instructions that are effective to reduce code size and cycle count. The new instructions implement the behavior of frequent primitives of the intermediate representation of GCC. To verify the proposed method, we construct a compiler, an assembler and a linker supporting the new instruction set. The experimental results show that new instruction can reduce the SPARC code size by 8.5% and cycle count 20% on the average.

References

[1] J. R. Hayes, M. E. Fraeman, R. L. Williams and T. Zaremba, "An Architecture for the Direct Execution of the Forth Programming Language," in *Proc. of the 2nd International Conference on ASPLOS*, 1987, pp. 42-49.

[2] D. R. Ditzel, H. R. McLellan and A. D. Berenbaum, "Design tradeoffs to support the C programming language in the CRISP microprocessor," in *Proc. of the 2nd International Conference on ASPLOS*, 1987, pp. 158-163.

[3] J. Hennesy and D. Patterson, *Compute Architecture: A Quantitative Approach*, Morgan Kauffman, San Francisco, 1996

[4] SPARC international, Inc., *The SPARC Architecture Manual*, Prentice Hall, Eglewood Cliffs, 1994.

[5] R. M. Stallman, *Using and Porting GNU CC for version 2.6*, Free Software Foundation Inc., 1996.

[6] Sun Microsystems, Inc., *Shade User's Manual*, 1999.

[7] A. V. Aho, M. R. Sethi, and J. D. Ullman, *Compiler—Principles, Techniques, and Tools*, Addison-Wesly, Reading, MA, 1986.