Compiler Generator for Hardware/Software Codesign

Hiroyuki Tomiyama Hiroki Akaboshi Hiroto Yasuura

Department of Information Systems,
Interdisciplinary Graduate School of Engineering Sciences,
Kyushu University

6–1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

E-mail: {tomiyama, akaboshi, yasuura}@is.kyushu-u.ac.jp

Abstract

Recently, due to the progress of CAD and device technologies, hardware/software codesign has attracted large attention. But lack of a compiler for the designed application specified processor makes it trouble to evaluate hardware and software together and to develop application programs. We have been developing a compiler generator for user designed processors. In this paper, we present the implementation and experimental results of the compiler generator. The compiler generator is based on tree-pattern matching and parameterized functions. Experimental results show that a compiler generated by the compiler generator have quality as good as an existing popular retargetable compiler, and that the compiler generator reduces description of processors to generate a compiler.

1 Introduction

Hardware/Software codesign is an effective approach to designing a digital system which contains micro processor units(MPU's). In order to optimize the system, it is effective to design an application specified processor for each application area, through analyzing application programs. When an architecture of an MPU is newly designed, a new compiler which translates application programs written in a high level language, like C language or Fortran, into target code of the MPU is required. But design of a compiler is too costly.

There exist some retargetable compilers[10] available in practical use, but they require human skill when changing the target into a new architecture. PEAS system[9] generates a core CPU and a C compiler from application programs automatically, but superset of the instruction set which the core CPU can implement is fixed. Instead of a compiler, a retargetable mapper which generates binary code through mapping algorithms to predefined hardware structure is proposed[8]. But it is not applicable to large algorithms because of its limitation on speed.

Our goal is to establish a method to generate a compiler to user defined processor architecture. We assume that all information of the architecture is extracted automatically from an HDL description of the design[3].

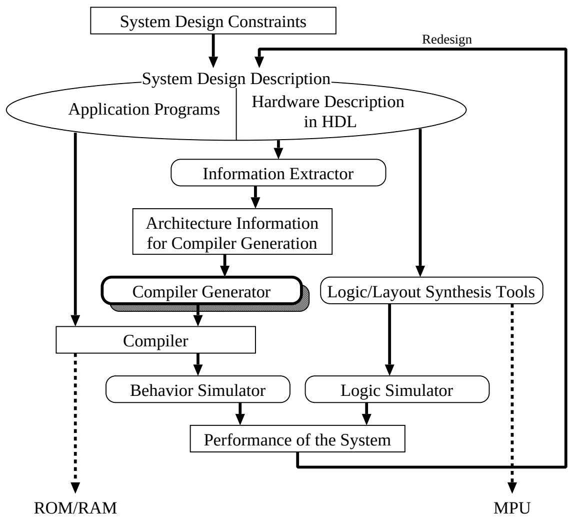

System Design Constraints
Redesign
System Design Description
Application Programs
Hardware Description in HDL
Information Extractor
Architecture Information for Compiler Generation
Compiler Generator
Logic/Layout Synthesis Tools
Compiler
Behavior Simulator
Logic Simulator
Performance of the System
ROM/RAM
MPU

Figure 1: Hardware/Software codesign methodology

In this paper, we present the implementation and experimental results of the compiler generator. The compiler generator generates a code generator from a formal architectural specification so that the system designer obtains a compiler easily even if he/she is not familiar with principles of compiler design. First, we introduce our hardware/software codesign methodology using the compiler generator. Secondly, we describe the prototype of the compiler generator and evaluate it. Finally, our future works are discussed.

2 Hardware/Software Codesign Using Compiler Generator

In Figure 1, we show a hardware/software codesign methodology we have proposed. A system designer describes the behavior of his/her MPU in HDL, and describes application programs in high level languages, like C language or Fortran. After compiling the hardware description written in HDL with logic/layout synthesis tools and performing logic simulation, the designer knows the performance of the MPU, such as clock frequency, chip area, power consumption, and so on. On the other hand, an information extractor extracts architecture information for compiler generation from the hardware description, and the compiler generator generates a compiler for the MPU automatically. The application programs can be compiled with the generated compiler and behavior simulation on the generated code can be performed. Therefore, the performance evaluation of both hardware and software can be achieved easily.

Thus, the compiler generator makes it easy for the designer to evaluate the performance of the designed system and provides a software development environment.

3 Compiler Generator

3.1 Target Architecture Model

To realize a compiler generator, several issues have to be considered.

1. Architectural specification from which a compiler is generated must be formal because we assume the architecture information is extracted from the hardware description written in HDL automatically.

2. Generated compilers must have quality as good as conventional compilers. Generally, quality of a compiler is estimated with the speed and size of the target code, compilation time, etc, but in this paper, we focus on the speed of the target code.

To simplify the problem, we assume that the target architecture is sequential one, namely no parallel operation including pipelining, and has some general purpose registers(GPR's).

3.2 Prototype of Compiler Generator

We assume a two-pass compiler, which consists of a front end and a code generator. Lexical analysis, parsing, semantics analysis, storage allocation, intermediate code generation and machine independent optimization are done in the front end, and the code generation and machine dependent optimization are done in code generator. Since tasks of the front end are machine independent in essence, we have firstly implemented a prototype of a code generator generator. The prototype generates a code generator which translates intermediate code into target code (See Figure 2).

First, we clarify the architecture information which the compiler generator requires. Main tasks of the code generator are instruction selection, register allocation, and machine dependent optimization. The architecture information required by the compiler generator is listed as follows.

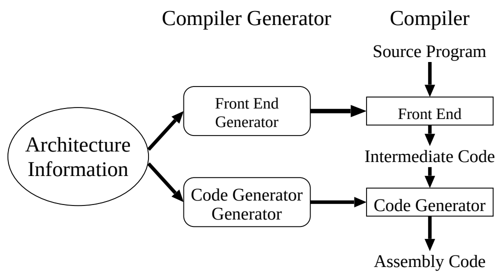


Figure 2: Compiler generator and generated compiler

- instruction set
  - behavior of each instruction
  - assembly notation of each instruction to be output
  - execution cycles of each instruction
  - instruction sequences on an entrance and an exit of a procedure
- register organization
  - the number of GPR's
  - register numbers of a stack pointer, a frame pointer, and a zero register if they exist.
  - the number of FPR's
- memory organization
  - wordsize
  - the number of storage units in a word

Next, instruction selection algorithm must be determined. Several algorithms have been developed. LR-like parsing algorithm[5] is typical one. But this algorithm sometimes fails to generate target code because of the fixed order of evaluating expressions. Tree-pattern matching is also well-known as a retargetable code generation algorithm. Aho et al combined tree-pattern matching with dynamic programming and developed *twig* system which is a tree-manipulation language and its translator[1]. We employ tree-pattern matching combined with dynamic programming as an instruction selection algorithm, and use *twig* as a part of the compiler generator.

Organization of the compiler generator is shown in Figure 3. A generated compiler consists of three phases, tree building, pattern matching, and register allocation. Tree construction function is machine independent. Tree construction are generated from the instruction set by the tree pattern generator, and twig specification is completed. Then, *twig* translates the twig specification into pattern matching function. Register allocation function is realized as a parameterized function, which is completed

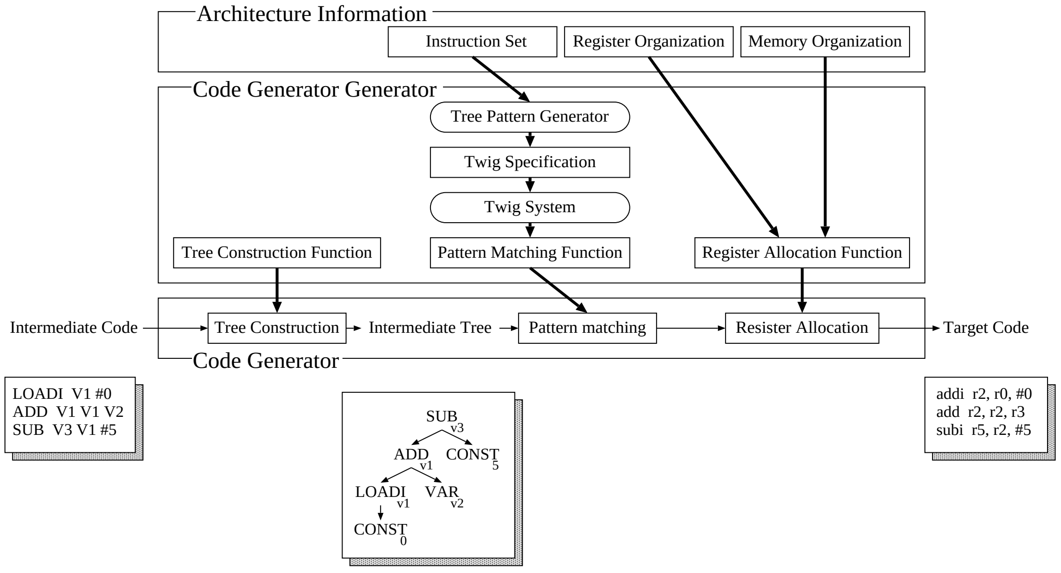
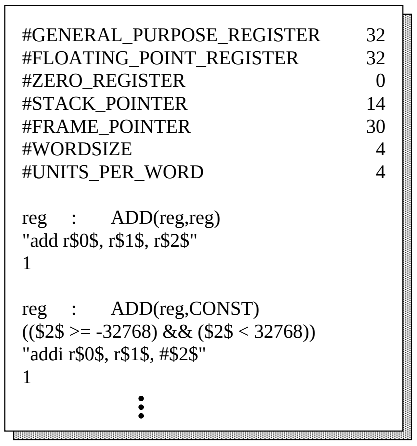


Figure 3: Prototype of the compiler generator



```
#GENERAL_PURPOSE_REGISTER      32
#FLOATING_POINT_REGISTER       32
#ZERO_REGISTER                  0
#STACK_POINTER                 14
#FRAME_POINTER                 30
#WORDSIZE                       4
#UNITS_PER_WORD                 4

reg    :    ADD(reg,reg)
"add r$0$, r$1$, r$2$"
1

reg    :    ADD(reg,CONST)
(($2$ >= -32768) && ($2$ < 32768))
"addi r$0$, r$1$, #$2$"
1
          ⋮
```

Figure 4: Example of an architectural specification

after parameters are added to it. Note that architecture information about register organization and memory organization can be completely parameterized.

We show an example of an architectural specification in Figure 4.

4 Experiments and Results

To evaluate quality of a compiler generated by the compiler generator, we have compared execution cycles of target code produced by a compiler generated by the compiler generator with target code produced by DLXcc which is based on GNU C Compiler(GCC) version 1.37.1. We use DLX[6] as a target architecture. Although the original DLX architecture is defined as a pipeline processor, we do not care about the optimization of target code depending on pipeline architecture. Architectural specification is given to the compiler generator manually, and translation from source programs into intermediate code is also manual work when the benchmark programs are compiled with the generated compiler. Optimizing option(–O) is used when the benchmark programs are compiled with DLXcc.

Table 1: Comparison of execution cycles

Programs	Ours	DLXcc
LLL1[†]	3,135	3,237
LLL2	3,190	3,516
LLL3	1,724	1,722
LLL4	1,637	1,405
LLL5	2,004	2,205
queen	29,844	26,637
hanoi	81,881	93,156
sieve	422,688	316,180

[†] Lawrence Livermore Loop

Table 1 shows that the generated compiler has quality as good as DLXcc. The reasons which caused these results are as follows.

- The generated compiler selected near-optimal instruction sequences.
- Because benchmark programs we used is small, the generated compiler could allocate variables to registers well, even if its power of register allocation is not as high as DLXcc.
- GCC version 1.37.1 on which DLXcc is based is not new.

Table 2: Size of the specifications of DLX instructions

	Ours	GCC
Instructions	67 instr.	87 instr.
Size	276 lines 5,414 bytes	1,613 lines 42,276 bytes

Table 3: Compiler generation time

	Ours	GCC
CPU time	31.3 sec	430.5 sec

There remains room to improve the compiler generator, particularly the register allocation method.

We have evaluated the facility of the compiler generator, too. The size of the specifications of DLX instructions required by the compiler generator and by GCC is shown in Table 2. The reason why the number of described instructions is different between the two is that our compiler generator do not support double precision floating point operations. Compiler generation time on Sparc Station 10 (SunOS 4.1.3, 64MB) is shown in Table 3. The generation time of GCC includes that of a front end, debugging supports, and so on, which are not implemented in our compiler generator. But we have confirmed that it takes over 186.4 seconds to generate a code generator of GCC, corresponding to the code generator generated by our compiler generator. These results show that the compiler generator requires much less tasks and time to generate a compiler.

5 Conclusion

We have introduced our hardware/software codesign methodology using the compiler generator and described the implementation and experimental results of the compiler generator.

In our current system, architectural specifications are given to the compiler generator manually. Study on an information extractor which extracts architecture information for compiler generation from hardware description written in HDL is now in progress[3]. We are sure that we will be able to connect the compiler generator to the information extractor and perform completely automatic compiler generation from a hardware description written in HDL in the near future.

A high level synthesis system for pipelined instruction set processor design which generates a reorder table for instruction scheduling has been studied[7]. Our compiler generator does not target parallel processing architecture. Extension to parallel processing architecture is also future work.

Acknowledgments

The authors would like to thank Prof. Kazuaki Murakami and the members of Yasuura laboratory of Kyushu University for their invaluable suggestions.

References

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”. *ACM Trans. on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addition-Wesley, 1986.

[3] H. Akaboshi, H. Tomiyama, and H. Yasuura. “Compiler Generation from Hardware Description Language”. In *Proc. of First Asian Pacific Conf. on Hardware Description Languages, Standards & Applications*, pages 76–78, 1993.

[4] M. Ganapathi, C. N. Fischer, and J. L. Hennessy. “Retargetable Compiler Code Generation”. *Computing Surveys*, 14(4):573–592, December 1982.

[5] R. S. Glanville and S. L. Graham. “A New Method for Compiler Code Generation”. In *Proc. of 5th ACM Symp. on Principles of Programming Languages*, pages 231–240, 1978.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[7] I.-J. Huang and A. M. Despain. “High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers”. In *Proc. of the 29th Design Automation Conf.*, pages 135–140, 1992.

[8] P. Marwedel. “Tree-Based Mapping of Algorithms to Predefined Structures”. In *Proc. of ICCAD-93*, pages 586–593, 1993.

[9] J. Sato, A. Y. Alomary, Y. Honma, T. Nakata, A. Shiomi, N. Hikichi, and M. Imai. “PEAS-I: A Hardware/Software Codesign System for ASIP Development”. *IEICE Trans. Fundamentals*, E77–A(3):483–491, March 1994.

[10] R. M. Stallman. *Using and Porting GNU CC for version 1.37.1*. Free Software Foundation, Inc., 1990.