A C Language Compiler Design in Comprehensive Experiment Methodology for Computer Science and Technology

Tian Wang, Yongtao Hao
College of Electronics Information & Engineering, Tongji University, Shanghai, China

Abstract - *The problems that exist in computer science today is too much attention has been paid to validate experiment, experiment content is mainly on a single aspect of technology and other issues. It is necessary to propose an experiment reform program for the computer science professional training. This program should cover the computer science foundation and core courses such as digital logic and digital IC, principle of computer organization, principle of microcomputer and interface technology, assembly programming, compiler theory and operating system. In this paper, a theory of C-compiler for the MIPS I instruction set processor is raised as an experiment reform program. The program is based on the LLVM project, and it is more innovative and practical compared to traditional compile-principle experiment.*

Keywords: Comprehensive experiment methodology, C language, Compiler porting, LLVM, GCC

1 Introduction

Low Level Virtual Machine (LLVM) is an open source project undertaken by the University of Illinois [1]. It provides support on the compiler, and can be used as a background for a variety of language compiler and used for compiler optimization, connection optimization, online compiler optimization and code generation. LLVM reuses the GCC as the frontend process tool of high-level language and provide its unique backend porting infrastructure to avoid the heavy workload of GCC backend porting [2].

2 LLVM Infrastructure

The architecture of LLVM compiler follows the logic compiler phasing rules[3], divided into the frontend of the high-level language processing, optimization of intermediate representation, backend code generator which is related to the target processor[4] (as shown in Figure 1).

LLVM uses GCC to analysis the source code written in high-level language and parse it into LLVM Intermediate Representation (IR). The latest version has supported for C, C + +, FORTRAN, JAVA. Moreover you can add new language support through frontend porting interface. The IR optimizer which is established on the LLVM virtual instruction set will process the IR generated in the previous step on standard scalar optimization, loop optimization, as well as inter-procedural optimization. This will generate optimized LLVM IR. The backend code generator contains instruction selection, register allocation, machine code optimization, code output and so on. The three parts above are relatively independent and optimization of one part does not affect the other two parts, so that we can improve the reuse of the module and reduce the unnecessary duplication of work [5].

3 LLVM Backend porting

Backend code generator will translate the LLVM IR, which is generated by optimizer, to assembly code for a specific processor. It contains two parts: the target code generator which is processor independent and backend transfer interface. Actual code is generated by the former process and to deal with it differently according to the description made by the latter part. This kind of division reduces the cost of the development of backend code generator greatly and makes the porting work reduced to the implementation of backend transplant interface for target processor.

LLVM backend porting interface is made up of abstract classes which describe the target processor architecture. Some of the classes are listed in table 1. For each specific processor, the first thing need to do is inheriting these classes, and then the corresponding feature attribute function should be implemented based on the new processor architecture. In this way, support for the new processor should be gained.

Table 1 Examples of the LLVM backend transplant interface

CLASS	USE
TargetMachine	Global description
TargetLowering	Description of IR conversion description
MRegisterInfo	Description of register
TargetInstrInfo	Description of instruction set
TargetFrameInfo	Description of frame stack layout
TargetSubtarget	Support of processor sub-series
TargetJITInfo	Support of processor JIT
……	……

TableGen In some cases, there may be a large number of records of description for the target processor needed to be

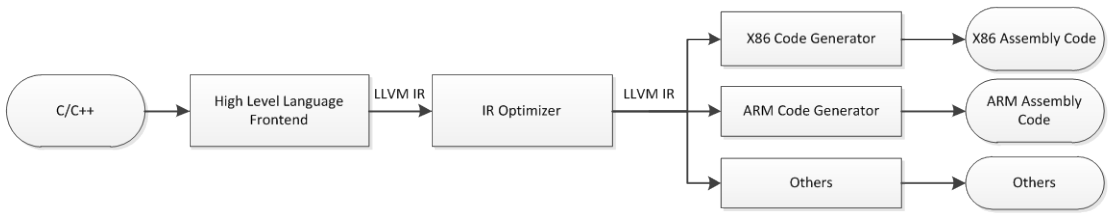

Figure 1 LLVM infrastructure

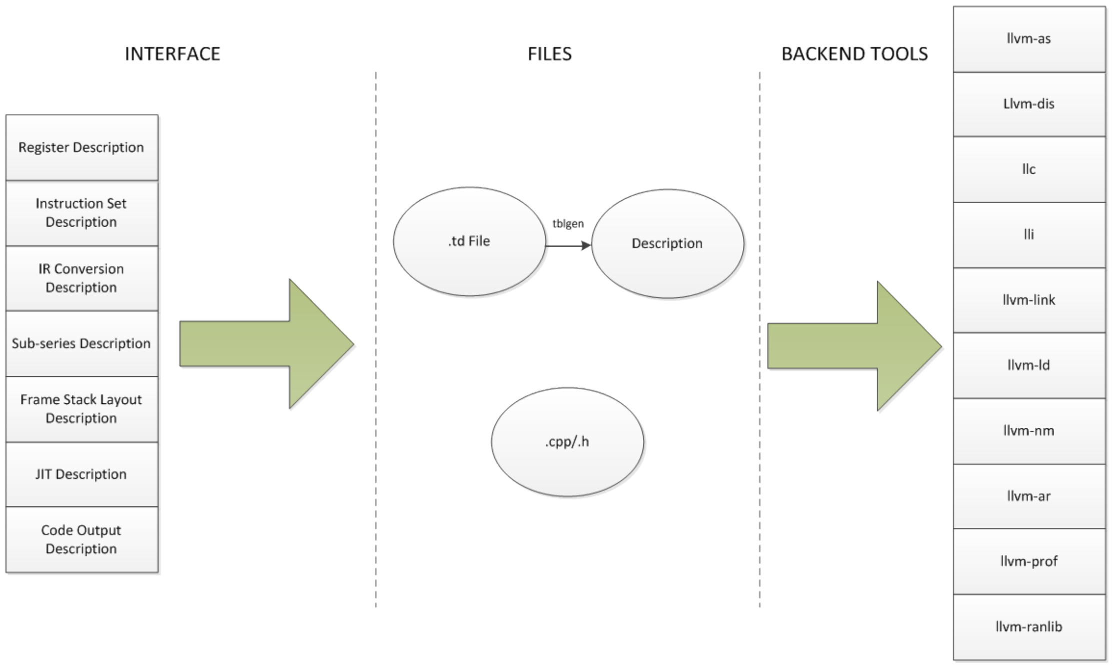

Figure 2 the LLVM backend transplantation structure

maintained, and they may have a lot in common. If these records are prepared artificially, it will spend a lot of time and will be very prone to error. To this end, LLVM provides a tool “TableGen” to reduce the workload of description. As long as users use .td file fits TableGen grammar rules to describe processor, tool tblgen can resolve it into C++ code. In this way, transplantation can be divided into two parts, using TableGen to describe the target processor and writing C++ code to complement it. Figure 2 above describes the structure of the LLVM backend porting [6].

Register description There are two aspects contained in the register description, TableGen description on the target processor and achieving class MRegisterInfo. That is to implementation these three files: XXXRegisterInfo.td 、 XXXRegisterInfo.h and XXXRegisterInfo.cpp. The first file describes properties of each register in processor, alias relationship between registers and register allocation scheme when programs are running. This is achieved by using the four records provided by TableGen. XXXRegisterInfo.h and XXXRegisterInfo.cpp need to inherit class MRegisterInfo and implement virtual function contained within the class. These virtual functions include providing the instruction that transmit the value in register to stack slot, providing the instruction that take the value from stack slot to register, providing instruction that copy register and so on.

Instruction set description The instruction set description also includes two aspects: TableGen description on the target processor’s instruction set and inheriting class TargetInstrInfo. The file XXXInstrInfo.td describes instruction set of the target processor, instruction features, addressing method of instructions, instruction operand, the instruction encoding method, the output format and the relation between instructions and virtual instruction set and so on. Document XXXInstrInfo.h and XXXInstrInfo.cpp need to inherit class TargetInstrInfo and implement the virtual functions in it. These function interfaces include the judgment of an instruction is a move instruction between registers[7], to determine whether an instruction is to read or write the stack slot command and so on.

IR conversion description IR conversion describes the way how LLVM IR converts to the assembly code of target processor. This can be taken down into three parts: the legalization of the operands, instruction matching options and other conversion description. As a matter of fact these can be achieved by fulfill documents XXXLowering.h and XXXLowering.cpp.

When LLVM IR's type and the type of target processor system are inconsistent, type conversion should be done. This progress is called legalization of the operands. Instruction set used by the LLVM IR is its virtual instruction set and this may not match the instruction used by target processor [8], so the conversion of instruction should be taken, too. Other conversion should be set for the shift size of the shift instruction type and target processor scheduling optimization options.

4 Conclusion

Currently, there has existed successful LLVM transplantation in the processor ARM and processor NiosII, thus porting LLVM to the processor based on MIPS I is feasible. Compared with traditional compile-principle experiment program, this program does not focus on the lexical analysis, syntax analysis and IR generation process, but on the assembly code generation. You will not be able to describe the target processor and accomplish porting of LLVM backend if you have not understood its structure and instruction set. This arrangement complements the traditional compiler theory experiment which is mainly on lexical analysis and syntax analysis and linking the relatively isolated compiler theory experiment to digital logic device design experiment, assembly language program design experiment, CPU design experiment which has been finished in the experiment reform program. This will help students to see these courses as a whole and to form their overall knowledge of computer science and technology.

5 References

[1] Terei A, Chakravarty T. An LLVM backend for GHC[C]. Proceedings of the Third ACM Haskell Symposium on Haskell. 2010:109-120.

[2] Xiaoxi Ren, Renfa Li, Kehuan Zhang, Yuanni Guo. A compiler technology based on retargetable method for embedded system[J]. Journal of Computer Applications 2004, 24(2):165-167.

[3] Lattner C, Adve V. Architecture for a next-generation GCC[C]. First Annual GCC Developers. 2003:121-132.

[4] S. Ren, N. Lu, W. Zhang, and Z. Pan. LLVM-infrastructure-based NIOS Ⅱ backend fast porting. Computer Applications and Software, vol. 28, 2011.

[5] F. Dong, Analysis of LLVM Compiler Infrastructure and Backend Porting For ARM[D]. vol. master: Shanghai Jiao Tong University, 2007

[6] F. Dong, Y. Fu. Backend porting for ARM based on LLVM infrastructure[J]. Information Technology. vol. 7,2007.

[7] G. Hadjiyiannis, S. Hanono, S. Devadas. ISDL: An instruction set description language for retargetability[C], In Proceedings of Design Automation Conference, pages 299-302, June 1997.

[8] Vikram Advd, Chris Lattner, Micheal Brukman, LLVA: A low-level virtual instruction set architecture[C]. Proc. of the 36th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-36), San Diego, CA, December 2003.