Investigation of flow graphs optimization methods used in optimizing compilers

Eugene Gavrin
SUAI
Eugene.A.Gavrin@hotmail.com

Abstract

This paper describes the flow graphs optimization techniques for use in a universal optimizing compiler. The following methods are comprehensive and can be used for optimizing compilation of most programming languages.

Index Terms: Compiler backend, optimizations, data flow graph, control flow graph.

I. Introduction

As our devices get smarter, they contain more semiconductors and draw more power. It’s also more difficult to write code that can efficiently utilize these multiple and sometimes multicore chips. A programmer can adapt its code to run on the specific hardware, or use a compiler that translates the code into chip-oriented assembly that the chip can process. But developing custom code for each platform is expensive, time-consuming and limited to the specific device. Most chipmakers had to develop compilers for each chip. While the process can take much time, the compiler can then be used for the chip in a variety of devices. But right after the new will be created this process needs to be repeated. To avoid this it is needed to create a universal compiler backend, which can be simply adapted to the specific architecture. This study is the first step towards developing the universal compiler backend.

II. Main part

The most compilers have the “classic” structure, and comprise the following parts. The source program is analyzed by the parser to produce an intermediate representation (IR) (abstract syntax tree or flow graphs), which is often called the abstract program [3]. Then the compiler performs a set of passes for optimization of intermediate representation. The abstract program is further translated by the code generator into a program in assembly language. A program in assembly language is very close to the target program, except that, instead of concrete cell addresses, it contains labels, each label representing some (yet) unknown address. The program in assembly language is then processed by the assembler, which replaces the entire label with concrete addresses, thereby producing the target machine code program. After that the compiler performs set of peephole optimizations to improve generated assembly.

Every new compiler is created mostly according to this structure. And its creation can be simplified by a standardized and extendable framework that implements some compiler stages: program analysis, IR optimizations, reconfigurable code generation and software correctness check.

A. Internal representation

As mentioned above, the compiler is divided into multiple steps, and to provide collaboration these steps is necessary to determine the internal and intermediate representation/language of the source program [5]. Intermediate representation is necessary in order to provide an interaction between multiple stages of the compiler - a concrete format is requires a separate research and will be made later. And the internal representation depends on the specific objectives of the particular stage. Currently is needed to select the representation suitable for a platform-independent of high-level program optimizations.

When parsing a program written in a high-level language the source representation is converted into a representation form suitable for the processing. The right internal representation simplifies the source code analysis and implementation of various code transformations. The choice of form of internal representation strongly affects the transformation methods, their efficiency and complexity, which can significantly affect the time of compilation.

The internal representation must meet certain requirements:

- *Ease of implementation of transformations;*
- *Explicit branching representation (cycles and conditions);*
- *Store the data dependency and control information;*
- *Store the useful properties of the original program;*

Most modern compilers for this purpose use the control flow graph that contains information about data dependence and control. In a control flow graph each node in the graph represents a basic statement block, i.e. a straight-line piece of code without any jumps or jump targets. Each statement block has a certain execution time, a necessary attribute for estimating the execution time of the program. Directed edges are used to represent jumps in the control flow, jump targets start a block, and jumps end a block. There are, in most representations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. And each possible execution path of the module has a corresponding path from the entry to the exit node of the graph.

B. Program transformations

The program transformation is any action that changes the internal representation of the program. Transformations can be structural, optimizing and debug. Structural are designed to enhance self-documenting and readability of the program. Optimizing are aimed to improve program effectiveness – to reduce the time of execution and memory consumption. And the transformations that build the debug version.

The optimizing transformations in the compiler are divided on three steps:

1. *Determine part of the program you need to optimize and determine the suitable transformation;*
2. *Check that the transformation does not change the result of program execution.*
3. *Transformation;*

The transformation is valid only if the original and the modified programs give the same output for different runs under the same input.

C. Platform independent optimizations

Optimization of the program is divided into two stages: a high-level global optimization, which is platform-independent and is universal for most programming languages, and low-level peephole optimization, which takes place after the generation of platform oriented assembly. The possible methods of platform-oriented optimizations will leave behind a side, and consider the optimization methods are universal for the any compiler.

Examples of optimizing transformations:

- *Constant folding;*
- *Sub-expression evaluation;*
- *Strength reduction;*
- *Null sequences;*
- *Combine operators;*
- *Loop optimizations;*
- *Branch Elimination;*

Constant folding is used to reduce redundancy. Constant values are computed at compile time and are passed to the program, which removes unnecessary arithmetic operations. There is also a subspecies called "Algebraic simplification", which eliminates unnecessary or incorrect arithmetic identities.

Direct conversion is replacement of slow operations with faster ones. For example, on many architectures, integer multiply instructions are slower than other instructions such as integer add and shift, and multiply expressions with power-of-two constant multiplicands and other bit patterns can be replaced with faster instructions. As well as integer divide is a relatively expensive instruction. Power-of-two integer modulus expressions can be replaced with conditional and shift instructions to avoid the divide and multiply and increase run-time performance.

Dead code elimination is reducing the size of the program by eliminating the sequence of operator blocks that is not achievable on any execution path in the program or that does not affect the program (e.g. dead stores). Such blocks can occur as a result of previous optimizations, debugging, or frequent changes in the program by many programmers.

An expression is a Common Sub Expression (CSE) if the expression was previously computed and the values of the operands have not changed since the previous computation. Recomputing the expression can be eliminated by using the value of the previous computation.

A loop containing a loop-invariant IF statement can be transformed into an IF statement containing two loops.

Loop overhead can be reduced by reducing the number of iterations and replicating the body of the loop.

Loop-invariant expressions can be hoisted out of loops, thus improving run-time performance by executing the expression only once rather than do it at each iteration.

Some same-conditional loops can be fused into one loop to reduce loop overhead and improve run-time performance. Although loop fusion reduces loop overhead, it does not always improve run-time performance – some architecture may provide better performance if two arrays are initialized in separate loops, rather than initializing both arrays simultaneously in one loop.

III. CONCLUSION

Described optimization methods should be applied to the original program until the program will no longer change its state. These methods can help create a software optimization and analysis framework that is going to be good basis for the reconfigurable compiler. It is understandable that it is possible to create a framework for creation of reconfigurable compiler. It is also possible to simplify compiler creation by providing general parts such as: parsing, analysis and optimization. The compiler remains two modules that can demand reconfiguration: parsing the source program and platform-oriented code generation. The task of parsing the source code is well known, as the methods of its solutions. The question remains open the creation reconfigurable code generator. Therefore, in future work, is planned to explore the methods of reconfigurable code generation and peephole optimizations currently exist and which ones might apply in practice.

REFERENCES

[1] Alfred Aho, Ravi Sethi, and Jeffrey Ullman, “Compilers: Principles, Techniques, and Tools”
[2] Robert Morgan, “Building an Optimizing Compiler”
[3] James Holmes, “Building Your Own Compiler with C++”
[4] Utpal Banerjee, “Loop Transformations for Restructuring Compilers”
[5] W. A. Barrett, "An ILOC Simulator", 2007, paraphrasing Keith Cooper and Linda Torczon, "Engineering a Compiler", Morgan Kaufmann, 2004. ISBN 1-55860-698-X.
[6] Microsoft Research, “Phoenix project”, http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx