International Journal of Engineering and Technical Research (IJETR)
ISSN: 2321-0869, Volume-2, Issue-9, September 2014

Compiler Design: Adaptive Compiler construction for Computing

Sadhana Gopal, Trishant Malik, Seema Devi

***Abstract*— In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called as the machine language. Since this is a tedious and error-prone process, most programming is done using a high-level programming language. A compiler translates or compiles a program written in a high-level language that is suitable for human programmers into the lower-level machine language that is required by the computers. During this process, the compiler will also spot and report the obvious programmer mistakes. This paper presents a compiler system for adaptive computing. This approach increases flexibility and usability in a way that allows porting the system to different targets with a minimal effort. Built on an existing design flow, we try to reach a new level of functioning by analyzing and partitioning C programs at highest possible description level. We show that analysis at this level is more efficient than on lower ones due to the exploitability of more expressive programming constructs. The improved analysis results can be combined with new SSA based algorithm for data path creation can lead to a higher solution quality of the final system configuration.**

***Index Terms*— adaptive compilation, compiler design, compilation sequences, searching good compilation sequence.**

I. INTRODUCTION

Traditionally, arithmetic performance of computing system is increased by faster or more processors. The term ‘adaption’ in computer science refers to a process, in which an interactive system or adaptive system adapts its behaviour to individual users based on information acquired about its user(s) and its environment. Adaptive systems accelerate programs by executing parts of the algorithm on adaptive hardware. Processors have grown more complex, with multiple functional units, exposed pipelines and myriad latencies that must be managed. In most cases, these computers execute code produced by compilers – a translator that consumes source code and produces equivalent for some target machine. These elements can be dynamically reconfigured during the program run. Some research projects in adaptive systems have already demonstrated the advantages. At the same time, the application of computing to new problems has created demand for compilers that optimize programs for new criteria, or new objective functions. For most modern processors, we can build optimizing compilers that produce efficient code for a single uniprocessor target. Compilers that are easier to retarget but produce less optimized code have been built so far. Such “retargetable” compilers are used in many situations where the economics cannot justify a large standalone compiler effort. What is yet to be developed is an economical way to produce high-quality compilers for a wide variety of target machines. Unfortunately, building such compilers is expensive, primarily because it requires years of effort by experts.

Manuscript received September 14, 2014.

Sadhana Gopal, CSE, MDU/DCE/, Gurgaon, INDIA, 9560086374
Trishant Malik, CSE, MDU/DCE/, Gurgaon, INDIA, 9811904225
Seema Devi, CSE, MDU/DCE/, Gurgaon, INDIA, 9560086374

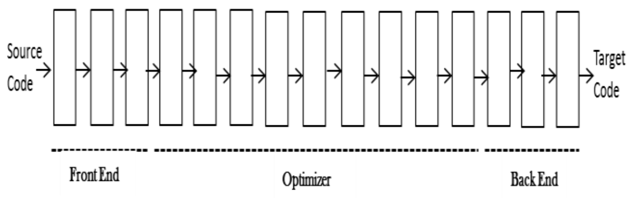

Fig.1 Classic Compiler Structure

This paper provides a framework for implementing optimizing compilers that will easily and automatically adapt their behaviour to the circumstances under which they operate, i.e., to different applications, to different target-machine performance parameters, to different sets of transformations and to different objective functions for optimization. The aim is to change the economics of producing high-quality compilers in a fundamental way and make it possible to build retargetable compilers that produce excellent code.

II. A New Structure for Compilers

One of the key challenges of future multi-core architectures is programmability and scalability. As compilers link the code written by programmers to the underlying parallel hardware, this cluster has a pivotal role to play. It will focus on versatility by adapting the user code to the ever changing underlying hardware. It will pursue a system wide perspective on adapting programs, and more generally workloads, to both short-term architecture variation, such as cache miss, and longer-term changes, such as the increasing number of processors available.
The choice of specific transformations and an order for their application play a major role in determining the effectiveness of an optimizing compiler. We call an ordered list of transformations a compilation sequence. Since the 1960s, complier writers have chosen compilation sequences in an ad hoc fashion, guided by experience and limited benchmarking. Efforts to find the best sequence have foundered due to the complexity of the problem. Transformations both create and suppress opportunities for other transformations. Different techniques for the same problem catch different subsets of the

available opportunities. Finally, combinations of techniques can achieve the same result as some single techniques.

Instead of translating directly into machine code, modern compilers translate to a machine independent intermediate code in order to enhance portability of the compiler and minimize design efforts. The intermediate language defines a virtual machine that can execute all programs written in the intermediate language. The intermediate code instructions are translated into equivalent machine code sequences by a code generator to create executable code. It is also possible to skip the generation of machine code by actually implementing the virtual machine in machine code. This virtual machine implementation is called an interpreter, because it reads in the intermediate code instructions one by one and after each read executes the equivalent machine code sequences of the read intermediate instruction directly. The use of intermediate code enhances portability of the compiler, because only the machine dependent code of the compiler itself needs to be ported to the target machine. The remainder of the compiler can be imported as intermediate code and then further processed by the ported code generator, thus producing the compiler software or directly executing the intermediate code on the code generator. The machine independent part can be developed and tested on another machine. This greatly reduces design efforts, because the, machine independent part needs to developed only once to create portable intermediate code.

Unfortunately, the best compilation sequences depends on many factors, including: 1) the specific details of the code being compiled, 2) the pool of available transformations, 3) the target machine performance and its performance parameters, 4) the particular aspect of the code that the user desires to improve(speed, page faults, power, space). Classic compilers try to address the second and third factors through design-time decisions, but ignore the first and last. This makes it difficult to predict the impact that changes in the compilation sequence will have on the compiled code. Today, we lack the knowledge to analytically predict the results of a particular sequence in a particular set of circumstances;

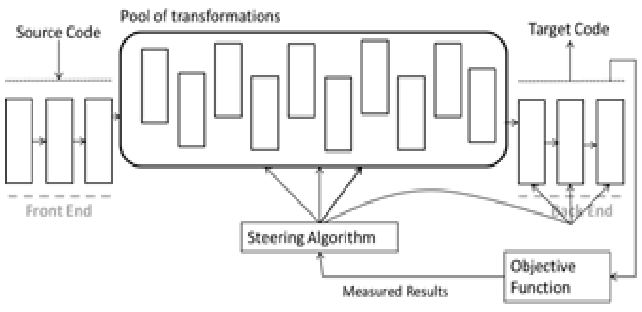

Fig.2 Structure of new compiler

this prevents a purely analytical process from deriving good code sequences.

In this paper, we describe a new approach to structuring compilation that promises to simplify the construction of high-quality optimizing compilers across a wide variation in all four of these factors. The system proposed in this paper, as shown in the figure, replaces the fixed-order optimizer with a pool of transformations, a steering algorithm, and an explicit, external objective function. The steering mechanism selects a compilation sequence and compiles the program with that sequence. The compiler evaluates the objective function on the resulting target-machine program. The measured results serve as input to the steering algorithm, allowing it to refine its choices and to explore the space of possible compilation sequences. Through repeated experiments, the steering algorithm discovers a compilation sequence that minimizes the objective function.

This approach addresses one of the fundamental challenges in the design and implementation of an optimizing compiler-choosing a specific set of transformations and an order of application for them- by computing the solutions. It relies on the speed of modern computers to replace the fixed-order compiler of the 1960s with a structure that adapts to new performance parameters, new input programs, new transformations, and new objective functions. It applies inexpensive cycles to solve a problem in compiler design that has defied both theory and practice for 40 years. It makes the compiler's objective function explicit, changeable, and multi-dimensional rather than implicit, fixed, and one-dimensional. The resulting compilers can optimize for a variety of objectives and for combinations of those objectives. We have done preliminary experiments using a particular search technique to find program-specific compilation sequences. To date, we have experimented with objective functions that optimize for code size, that optimize for speed, and that optimize for a property related to power consumption.

III. SEARCHING FOR COMPILATION SEQUENCES:

Most effective compilers include ten to twenty transformations currently, drawn from the hundreds that have been proposed in the literature. Picking the best compilation sequence for a specific program and a given objective function is hard: 1) there is little theoretical understanding of the effect of particular compilation sequences on the external objective function, and 2) the space of compilation is too large for approaches relying on exhaustive search. Most compilers offer a small number of compilation sequences (-01,-02,-03,…) discovered manually by designers. If none of these sequences is good fit to the application or the user's real performance goals, the user has no recourse.

Picking compilation sequences is an instance of a family of combinatorial problem called sequential decision-making problems. These problems have the following properties:

- Solving a problem requires making a sequence of decisions.
- The effect or outcome of each decision is a function of decisions made in the past as well as the other random factors not entirely within the decision-maker's control.
- A decision made at a given point in time alters the set of choices for the future. At each step, the future impact of a decision must be considered.
- The objective function depends in a complex way on the interactions between the individual decisions and their stochastic outcomes.

The problem of finding compilation sequences for specific circumstances is such a problem. The travelling salesman problem (TSP) and other discrete combinatorial optimization problems are also members of this problem class. The standard approach to solving these problems a=uses deterministic or stochastic dynamic programming. Since traditional dynamic programming implementations need excessive amounts of space, complete search algorithms, for example branch and bound algorithms for TSP, are used. Complete search algorithms guarantee a globally optimal solution. However, they are only practical for problems where effective pruning techniques are known. Unfortunately, too little is known about picking compilation sequences to enable early pruning.

Searching a good optimization sequence for a program in the optimization sequence space is like searching for a needle in a haystack. Instead of solving this problem on a per program basis, compiler writes construct good optimization sequences through experience and experimentation on benchmark programs. The constructed sequences are associated with default optimization options like –o1, -o2 and –o32. These sequences may be globally optimal with respect to the program spaces, but are sub-optimal for the individual programs. The simple reason being, what a good sequence for one program is may not be good for another. So looking for a universal good sequence is a futile exercise. An alternate viable approach is to build a set of few good sequences, so that for every class of programs there is a good optimization sequence in the sequence set catering to that class. Then given a new program we can choose the best sequence by trying out all the sequence from the good sequences set. This approach completely bypasses the program classification problem. Using the LLVM compiler framework [Lattner and Adve 2004] we construct a good sequence set.

IV. Conclusion

The adaptive compilers that result from this work will allow researchers and complier writers to explore the space of compilation sequences and the impact of those on code quality. To make these ideas useful in practice, we must design mechanisms that use the results of full-fledged adaptive compilations in limited time compiles. To build efficient production compilers from this configurable base will require additional research and implementation.

References

[1] John Bakus. The history of Fortran I, II and III.
[2] Keith D. Cooper and Tim Harvey. A study estimated name transitions in Fortran codes. Technical report in preparation, available on web at http://softlib.rice.edu/MSCP/Publications.html, April 2001.
[3] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In proceedings of the 1999 workshop of languages, compilers and tools for embedded systems, May 1999.
[4] Nicolas G. Fournier. Enhancement of an evolutionary optimizing compiler. Master’s thesis, Department of computer science, University of Manchester, September 1999.
[5] Bacon, D. F. Graham, S.L., Sharp O.J., Compiler Transformations for High-performance Computing, ACM Computing Surveys, 194.
[6] LLVM. 2012. http://llvm.org/docs/Passes.html