An Interactive Compiler Development System

Gary S. Tyson, Robert J. Shaw and Matthew K. Farrens
Division of Computer Science
University of California, Davis, CA 95616
email: tyson@cs.ucdavis.edu, tel: (916) 752-7004

Abstract

An interactive compilation environment has been developed to facilitate the rapid prototyping of machine dependent code optimization strategies for the Decoupled Processor Design project under development at University of California, Davis. This paper describes an interactive graphical optimizer based on the Tcl and Tk libraries. An overview of the optimizer is presented along with some motivation for the unique features it provides.

1. Introduction

The development of high performance architectures requires considerable interaction between the architectural specification and the machine specific optimizations performed to exploit the capabilities of the architecture. These optimizations often expand on more general techniques found in current compilers [Stal]. However, few tools exist to aid in the integration of new and existing optimization techniques. We have developed an *Interactive Graphical Optimizer (IaGO)* to facilitate the construction of a high performance code optimizer for new target architectures. This system allows for much greater control of the application of optimization techniques by incorporating a Tcl based script language into the code optimizer. In addition, the use of Tk to generate an interactive interface between the compiler developer and the internals of the optimizer allows for new code optimization strategies to be applied *on the fly*.

2. Motivation

New high performance architectures are currently being developed at numerous university and corporate research centers. At UC Davis, we are investigating new architectural approaches that exploit the implicit *Instruction-Level Parallelism (ILP)* found in conventional sequential programs (in our case, **C** source programs). The increased capacity found in these new architectures requires more sophistication on the part of the compiler to realize an improvement in performance. Generally, the more complex the architecture, the less applicable current compiler technology becomes in the generation of efficient code. We have developed a set of tools to facilitate the design and analysis of the *Multiple Instruction Stream Computer (MISC)* [TyFP92] architecture and simplify the construction of new optimization strategies suited to the unique capabilities of this architecture. This paper briefly discusses one of these tools, IaGO, which provides an interactive compilation environment used to develop prototype code optimization strategies for MISC and other new architectures.

The MISC architecture uses multiple asynchronous processing elements to separate a program into instruction streams that can be executed in parallel. Unlike other MIMD[1] architectures, MISC has been designed to separate a task into multiple, finely interleaved instruction streams which cooperate to execute a sequential task; this is the same ILP exploited by Superscalar architectures such as DEC's Alpha processor [Site93]. The partitioning of the task requires the compiler to identify both independent and dependent operations and to assign them to different processing elements. The separation of instructions to exploit ILP is a relatively new strategy and compiler support is unavailable. IaGO allows new optimization strategies to be attempted with minimal delay and with much greater flexibility than current optimizers — which generally use command line arguments to specify which optimization techniques should be tried. Among the questions that we wish to study are optimal strategies for instruction stream separation, tradeoffs in register allocation and instruction scheduling, and methods for hiding operational latencies by controlling the asynchronous entry of processing elements into basic blocks.

IaGO provides two key advantages over alternative compiler models. First, the application and ordering of optimization methods can be specified by a command script, allowing alternative schemes to be attempted without regenerating the compiler. This is important because the relationship among code transformations is complex and the effects of architectural dependencies can exclude particular transformations or particular orderings of optimizations. Second, with the use of Tk, the compiler developer can interact with the internals of the optimizer during the compilation process. Code can be *hand optimized* by allowing the developer to manipulate the internal representation of the program (e.g. rewrite the intermediate language program description or modify dataflow information). This allows

[1] Multiple Instruction / Multiple Data

new optimization strategies to be evaluated without the necessity of coding them in C or as an optimization script.

3. Compiler Overview

Once we decided to develop a compiler model for MISC, a study was made of existing compilers and the *very portable C compiler (vpcc)* [BeDa91] was chosen as the base model for IaGO. The design of vpcc, ongoing at the University of Virginia, is an extension of the portable C compiler developed at Bell Labs. The vpcc compiler is separated into two phases: the parser or front-end and the code optimizer or back-end (see figure 1).

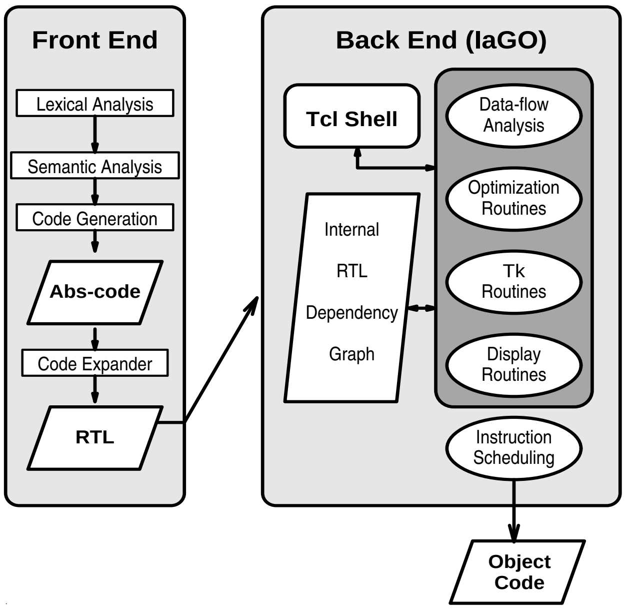

Figure 1: Overview of the vpcc/IaGO compiler

The front end of the compiler parses C source code and generates naive (but correct) code for a simple abstract machine (**Abs-code**). The code expander translates the abstract machine code into *Register Transfer Lists (RTLs)*; an RTL is a machine specific representation for the target machine specified in a machine independent form. This independent form allows dataflow analysis and many optimization routines to operate in a machine independent manner. Once an RTL description of the program is generated, it is written to a file and the optimization phase is initiated. A compilation driver program is responsible for coordinating the execution of the parts of the compiler — including pre-processing, assembly and linking operations.

The second phase of the vpcc compiler has been modified to support IaGO. IaGO consists of a Tcl interpreted shell, a set of routines to perform dataflow analysis, code optimization and graphical display. When IaGO is invoked during the compilation process, a shell script is provided to the interpreter in addition to any command line arguments (provided by the driver program). Normal operation of IaGO starts with a series of commands (specified by the script file) to load the intermediate RTL description of the program, perform dataflow analysis to construct a dependency graph and apply whichever optimization transformations to the code are specified by the script. Once optimization is complete, instruction scheduling is initiated to generate a final object (assembly code) listing of the program. The driver program can continue with assembly and linking phases if requested.

4. Optimization Script Language

The first component of the IaGO system is an IaGO command shell. This command shell is simply a Tcl interpreter augmented with optimization and display routines. Command line arguments received from the compilation driver determine the location of the IaGO script controlling the optimization process. This will usually involve specifying an IaGO command script on the vpcc command line. Application of optimization routines is controlled by the script; RTL files will be opened, contents read, optimizations performed and assembly code generated by invoking various Tcl and IaGO procedures. An interactive shell can also be specified for simple textual interaction between IaGO and the developer.

Most IaGO routines operate on a global RTL dependency graph; all code translations maintain the same internal format, so there is no required ordering of optimization operations. This provides the command script with (almost) complete freedom in scheduling code translations. In addition to the optimization routines, IaGO registers several data conversion routines to allow access to the internal data structures by the command interpreter. This allows the command scripts to access internal structures as shell variables and to determine control flow of the optimization process accordingly. The IaGO command language then has the full programmability found in the Tcl language. Iteration can be performed to control the application of any of the optimization (or dataflow analysis) routines. More aggressive scheduling of transformations can then be attempted without sacrificing correctness of target code or determination (guaranteed compiler optimization termination).

5. Interactive Optimization

Another useful capability of the IaGO system is its ability to interact with the compiler developer to generate more efficient code or to develop new optimization techniques. A menu driven graphical interface can be created (from the IaGO command shell) to provide detailed information about the internal state of the optimization process and to accept commands input by the developer.

When using this graphical interface, the structure of the program is viewed as a set of basic blocks, specified in RTL format, displayed in Tk listboxes. Control flow is displayed as directed arcs between the basic block (on a Tk canvas). This representation of control flow can be augmented with information regarding data dependencies, register usage or higher level semantic structures such as loops. Many of the display characteristics can be specified by the command shell allowing more of the internal representation to be viewed.

The primary mechanisms for direct manipulation of the compilation process are Tk menus and a text editor. Dataflow modifications are made by manipulating the items on the canvas. RTLs can be directly manipulated by invoking a text editor on the RTL representation of the code in a basic block. This interaction between the compiler developer and the optimization routines allows for more sophisticated transformations to be applied. Compilers have little difficulty with applying global transformations (e.g. global register allocation or code motion) across the entire scope of a function. People have far more difficulty applying these types of transformations. However, people are proficient at determining semantic information about the application. This often allows human intervention to avoid overly conservative scheduling decisions by the optimizer. An example of this is the analysis required to guarantee that no memory (aliasing) hazards exist in the schedule. Often a person can provide this guarantee by examining the application when the compiler cannot guarantee this with a detailed analysis of the low level semantics found in the intermediate code.

The ability to quickly evaluate new optimization strategies by interactively applying transformations provides the compiler developer with a powerful tool for studying the underlying characteristics of advanced architectures.

6. Future Research

Although we have presented IaGO as a compiler writer's tool, it can be applied to many other programming projects with only modest alterations. For instance, the multiprocessing community accepts that a truly general-purpose parallel architecture will never exist, and consequently, the programmers of such machines must become intimately familiar with their particular architectures. If they do not, they fail to harness the full power which the machine has to offer. Even today, in supercomputer centers such as the Lawrence Livermore National Laboratory, efforts to fine tune applications consume a large portion of the programming professionals. For these individuals, a tool such as IaGO would allow much greater interaction between the compilation model and themselves. Rather than a one-way conversation with the compiler through compiler directives embedded in some parallel dialect of Fortran or C, IaGO would allow the programmer to interact with the optimization process directly. The compiler performs the computations it does best (e.g. dependency analysis and global register allocation), while the programmer provides the advice that the software can not determine with certainty (the absense of pointer hazards and function side-effects). IaGO allows the programmer to communicate higher-level aspects of the program design to the compiler — aspects which are all but invisible at the level of basic blocks and RTLs. Again, the fact that detailed architectural knowledge is required to use IaGO effectively in this way is not a drawback because such knowledge is needed anyway by the application tuners working on high-performance platforms.

Because of the ease of programming that Tcl provides, several additions to IaGO's displayed information are possible, all of which serve to enrich the nature of this bidirectional programmer/compiler interaction. An obvious extension is to include profiling information so that the programmer sees clearly which basic blocks are crucial to high performance. Similarly, common compiler notions such as live ranges, natural loops, and use-def chains are all easily incorporated into IaGO, allowing the programmer to perform high-level code reorganization to promote the compiler's skill at code motion and register allocation.

IaGO has displayed great benefit in the development of new optimization strategies for the MISC processor. We believe that the capabilities found in an interactive compilation environment can be applied to a more general field of programming. Once the development of IaGO has matured in its existing configuration, we wish to port it to a compiler with greater availability such as gcc.

References

[BeDa91] M. E. Benitez and J. W. Davidson, “Code Generation for Streaming: an Access/Execute Mechanism”, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA (April 8-11, 1991), pp. 132-141.

[Site93] R. L. Sites, “Alpha AXP Architecture”, *Communications of the ACM,* vol. 36, no. 2 (February, 1993), pp. 33-44.

[Stal] R. M. Stallman, *Using and Porting GNU CC*, Free Software Foundation, Inc. 1991.

[TyFP92] G. Tyson, M. Farrens and A. Pleszkun, “MISC: A Multiple Instruction Stream Computer”, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Portland, Oregon (December 1-4, 1992), pp. 193-196.