ABSTRACT

This paper presents that Compiler infrastructures that support experimental research are crucial to the advancement of high-performance computing and in order to develop such infrastructure we require huge investment. Therefore SUIF compiler are being generated and convert the SUIF compiler into a powerful, flexible system. SUIF consists of documented Kernel and toolkit of compiler passes and it is very expensive, if not impossible, to develop a fully functional compiler platform that embodies all of the known program analyses and optimization techniques. This is especially true in the arena of compiler research for high-performance systems, where both conventional data flow optimizations and high-level transformations are necessary to improve parallelism and memory hierarchy performance.

KEYWORDS: INFRASTRUCTURES, KERNEL, COMPILER PASSES, PARALLELIS

1. INTRODUCTION

Machine SUIF is a flexible, extensible, and easily-understood infrastructure for constructing compiler back ends. The optimization and analysis passes, should be coded in a way that makes them as independent of the compiler environment and compilation targets as possible. In particular, the Machine-SUIF distribution contains a working compiler based on the Stanford SUIF compiler infrastructure. This compiler is capable of producing optimized code for machines based on the Alpha or x86 architectures. However, the analyses and optimizations distributed in Machine SUIF do not directly reference any SUIF constructs or embed constants from any target machine.

2. THE KERNEL DESIGN

The SUIF Kernel performs various functions:

- It defines the intermediate representation of programs. The level of representation for programs in a parallelizing compiler is a crucial element of the compiler design. At the other extreme, many parallelizing compilers are source-to-source translators and their analyses and optimizations work directly on abstract syntax trees. Furthermore, all the compiler algorithms must be able to handle a
rich set of source constructs, thus rendering the development of such algorithms are more complicated.

The symbol tables in a SUIF program hold detailed symbol and type information. Information is complete enough to translate SUIF back to legal and high-level C code. The system keeps sufficient information about Fortran arrays and common blocks to enable full interprocedural analysis.

- It provides functions to access and manipulate the intermediate representation. The SUIF library defines a C++ class for each element of the program representation, allowing us to provide interfaces to the data structures that data hide the underlying details. We gain type safety and modularity by putting the shared features in a base class and then deriving subclasses for each variant. Besides the basic implementation of the SUIF data structures, the kernel provides numerous features to make the system easy to use. It defines a variety of generic data structures including hash tables, extensible arrays, and several kinds of linked lists. The functions that we have found to be needed most frequently for each class are often included as methods.

- It structures the interface between compiler passes. Compiler passes interact with one another either by updating the SUIF representation directly or by adding annotations to various elements of the program. Users define each kind of annotation with a particular structure so that the definitions of the annotations serve as definitions of the interface between passes. We can thus easily replace an existing pass with another pass that generates the same annotations. The annotation facility also encourages experimentation with new abstractions. Adding new information in annotations does not affect the rest of the system, making it easy to investigate various alternatives.

3. SUIF Compiler Toolkit

The SUIF compiler toolkit consists of:

- Fortran Front Ends

  We do not have a Fortran front end that directly translates Fortran into SUIF. Instead, we use AT&T to translate Fortran 77 into C, followed by our C front end to convert the C programs into SUIF. The ideal solution would be to build a Fortran front end for SUIF, but we have not yet found time to work on that. In the meantime, the current solution works for most benchmarks and is sufficient for research purposes.

- A Loop Level Parallelism and Locality Optimizer

  The SUIF paralleliser is made up of many different compiler passes. First, a number of scalar optimizations help to expose parallelism. These include constant propagation, forward propagation, induction variable detection, constant folding, and scalar privatization analysis. Next, unimodular loop transformations guided by array dependence analysis restructure the code to optimize for both parallelism and locality. Finally, the parallel code generator produces parallel code with calls to the parallel run-time library. The algorithm operates on loops with either distance or direction vectors. The unimodular transformations are applicable to perfectly nested
loops, as well as imperfectly nested loops if the code outside the innermost loops consists of simple statements and not loops. The mechanisms for performing these loop transformations are provided in a library that other researchers can use to implement different loop transformation policies.

- **An Optimizing MIPS Back End**

Many compiler research projects need access to an optimizing back end, even if they do not directly deal with scalar optimization or code generation. To support such work, the initial release of the SUIF toolkit includes a set of conventional data-flow optimizations and a MIPS code generator.

- **Compiler Development Tools**

The compiler toolkit includes many facilities to aid in the compiler development process. Several SUIF passes deal extensively with integer matrices and systems of linear inequalities, so we have extracted a set of common mathematical functions into a mathematics library. Included in the library is a Fourier-Motzkin elimination algorithm for solving systems of real-valued inequalities, extensions to Fourier Motzkin to solve for integer solutions, as well as extensions to work with a class of linear inequalities that have symbolic coefficients. The library also supports linear algebra. We have developed an interactive interface to this mathematics library, the Linear Inequality Calculator (LIC). This tool allows a compiler developer to easily test out new algorithms on examples and has also been used in a Stanford compiler course to aid students in learning about data dependence analysis and parallel code generation. High-level program transformations often need to generate large amounts of SUIF code. For example, the parallel code generator must create upper and lower bound expressions to schedule the iterations of a parallel loop. Constructing such expressions directly from low-level SUIF objects requires many tedious and error-prone operations. To simplify this process, the “builder” library translates C-like statements into SUIF code. The builder interface also insulates the user from future changes to the internal program representation.

- **A Simplified SUIF for Instructional Use**

This interface is tailored specifically for implementing scalar data-flow analyses and optimizations. It only exposes to the students the information relevant for these problems. However, because Simple-SUIF retains all the information in the SUIF files, students are able to test their projects in the context of a fully functional C compiler. This system has been used in a compiler optimization course at Stanford. Students were able to use Simple-SUIF to develop several of their own optimizations within a quarter-based course.
CONCLUSION

SUIF is a growing system. We are continually developing and experimenting with new compiler techniques. As new passes are completed, we will add them to the toolkit, potentially replacing less powerful passes. We view SUIF as a bootstrapping system. The kernel will retain the same basic interface, possibly with extensions to support new areas of research. The various components in the toolkit, however, may one day all be replaced with passes embodying newer compilation techniques and better engineering designs. Our goal is to establish an interface between compiler passes so that other researchers can collaborate in developing the SUIF infrastructure. We thus invite you to use SUIF and welcome your contributions to the system.

REFERENCES


