The SUIF Compiler System: A Parallelizing and Optimizing Research Compiler

Robert P. Wilson, Robert S. French, Christopher S. Wilson,
Saman P. Amarasinghe, Jennifer M. Anderson,
Steve W. K. Tjiang, Shih-Wei Liao,
Chau-Wen Tseng, Mary W. Hall,
Monica S. Lam, and John L. Hennessy

Technical Report CSL-TR-94-620

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University, CA 94305-4055

Abstract

Compiler infrastructures that support experimental research are crucial to the advancement of high-performance computing. New compiler technology must be implemented and evaluated in the context of a complete compiler, but developing such an infrastructure requires a huge investment in time and resources. We have spent a number of years building the SUIF compiler into a powerful, flexible system, and we would now like to share the results of our efforts.

SUIF consists of a small, clearly documented *kernel* and a *toolkit* of compiler passes built on top of the kernel. The kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes. The toolkit currently includes C and Fortran front ends, a loop-level parallelism and locality optimizer, an optimizing MIPS back end, a set of compiler development tools, and support for instructional use.

Although we do not expect SUIF to be suitable for everyone, we think it may be useful for many other researchers. We thus invite you to use SUIF and welcome your contributions to this infrastructure. Directions for obtaining the SUIF software are included at the end of this paper.

1 Introduction

The compiler research community has a great need for compiler infrastructures on which new technology can be implemented and evaluated. Besides the basic requirements for various parsers and code generators, a good infrastructure must include all the important program analyses and optimizations. 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. Developing such a fully functional infrastructure is a huge investment of time and resources.

Because independently developing an entire infrastructure is prohibitively expensive, compiler researchers would benefit greatly from sharing investments in infrastructure development. Toward that end, we are now making our SUIF (Stanford University Intermediate Format) compiler system available to others. We have developed SUIF as a platform for our research on compiler techniques for high-performance machines. It is powerful, modular, flexible, clearly documented, and complete enough to compile large benchmark programs. While SUIF is probably not suitable for everyone, we believe that it could be useful to many.

Our group has successfully used SUIF to perform research on topics including scalar optimizations, array data dependence analysis, loop transformations for both locality and parallelism, software prefetching, and instruction scheduling[1]. Ongoing research projects using SUIF include global data and computation decomposition for both shared and distributed address space machines [2], communication optimizations for distributed address space machines [1], array privatization [7], interprocedural parallelization [5], efficient pointer analysis, and optimization of Verilog simulations. SUIF has also been used for courses on compiler optimizations at Stanford. This paper only presents the base infrastructure of SUIF and does not include our latest research. However, as our work progresses, we will continue to incorporate our results in the compiler.

Our goal with SUIF is to construct a base system that can support collaborative research and development efforts. Thus, we have structured the compiler as a small *kernel* plus a *toolkit* consisting of various compilation analyses and optimizations built using the kernel. The kernel defines the intermediate representation and the interface between passes of the compiler. This interface is always the same so that the passes in the toolkit can easily be enhanced, replaced, or rearranged. This is ideal for collaborative work and has also made it practical for us to develop a usable system.

All program information necessary to implement scalar and parallel compiler optimizations is easily available from the SUIF kernel. The intermediate program representation is a hierarchy of data structures defined in an object-oriented class library. This intermediate representation retains almost all the high-level information from the source code. Accessing and manipulating the data structures are generally straightforward due to the modular design of the kernel. We also provide extensive documentation for the kernel.

The SUIF toolkit contains a variety of compiler passes. Fortran and ANSI C front ends are available to translate source programs into SUIF. The system includes a parallelizer that can automatically find parallel loops and generate parallelized code. A SUIF-to-C translator allows us to compile the parallelized code on any platform to which our parallel run-time library has been ported. Besides generating C, our system can also directly produce optimized MIPS code. The system provides many features to support parallelization: data dependence analysis, reduction recognition, a set of symbolic analyses to improve the detection of parallelism, and unimodular transformations to increase parallelism and locality. Scalar optimizations such as partial redundancy elimination and register allocation are also included.

This paper is an introduction to the SUIF system. We first describe the SUIF kernel in section 2 by giving an overview of the intermediate representation along with some highlights of the kernel implementation. We then proceed to briefly outline some of the components of the toolkit in section 3.

2 The Kernel Design

The SUIF kernel performs three major functions:

- *It defines the intermediate representation of programs.* This representation supports both high-level program-restructuring transformations as well as low-level analyses and optimizations.

- *It provides functions to access and manipulate the intermediate representation.* Hiding the low-level details of the implementation makes the system easier to use and helps maintain compatibility if the representation is changed.

- *It structures the interface between compiler passes.* SUIF passes are separate programs that communicate via files. The format of these files is the same for all stages of a compilation. The system supports experimentation by allowing user-defined data in annotations.

In the following subsections, we expand on each of these aspects of the kernel.

[1]The current SUIF system is a major revision of an earlier version [12]. Some of our previous work has not yet been ported to the new system and is not available at this time. We hope to include it in a future release.

2.1 Program Intermediate Format

The level of representation for programs in a parallelizing compiler is a crucial element of the compiler design. Traditional compilers for uniprocessors generally use program representations that are too low-level for parallelization. For example, extracting data dependence information is difficult if the array accesses are expanded into arithmetic address calculations. At the other extreme, many parallelizing compilers are source-to-source translators and their analyses and optimizations work directly on abstract syntax trees. While these abstract syntax trees fully retain the high-level language semantics, they are language-specific and cannot easily be adapted to other languages. For example, extending a Fortran 77 compiler to accept Fortran 90 programs would be difficult. Furthermore, all the compiler algorithms must be able to handle a rich set of source constructs, thus rendering the development of such algorithms more complicated.

Our intermediate format is a mixed-level program representation. Besides the conventional low-level operations, it includes three high-level constructs: loops, conditional statements, and array accesses. The loop and conditional representations are similar to abstract syntax trees but are language-independent. These constructs capture all the high-level information necessary for parallelization. This approach also reduces the many different ways of expressing the same information to a canonical form, thus simplifying the design of the analyzers and optimizers.

SUIF instructions are the primary components of each procedure body. High-level compiler passes typically group the instructions into expression trees. Most instructions perform simple RISC-like operations, but a few more complex instructions are used to avoid losing information about array accesses, procedure calls, and switch statements. Unstructured or irregular control flow is simply represented by lists of instructions that include branch and jump operations.

As a compilation progresses to the back end optimizations and code generation, the high-level constructs are no longer needed and are expanded to low-level operations. The result of this expansion is a simple list of instructions for each procedure. Information from the high-level analyses can easily be carried down to the low-level representation via annotations. For example, data dependence information can be passed down for instruction scheduling by annotating the load and store instructions.

The symbol tables in a SUIF program hold detailed symbol and type information. The 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.

Support for interprocedural analysis and optimization is also built into the intermediate representation. Since most significant programs have multiple source files, interprocedural algorithms must be able to deal with references across files. We have addressed this by explicitly including the source files at the top level of the intermediate representation. All source files share the same global symbol table so that references to global symbols are the same from any file.

2.2 An Object-Oriented Implementation

The kernel provides an object-oriented implementation of the SUIF intermediate format. This SUIF library defines a C++ class for each element of the program representation, allowing us to provide interfaces to the data structures that hide the underlying details. We have found that inheritance and dynamic binding work particularly well in this context. Many classes have basically the same components with a few added features for each subclass. For example, variables, labels, and procedures are all symbols, but although they share the same basic interface, only procedure symbols have bodies of code associated with them. By putting the shared features in a base class and then deriving subclasses for each variant, we gain type safety and modularity.

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. These include methods to traverse the program representation, duplicate regions of code, and create new instructions and symbols. The low-level details of reading and writing SUIF files are also handled entirely by the library.

2.3 Interface Between Compilation Passes

The SUIF toolkit consists of a set of compiler passes implemented as separate programs. Each pass typically performs a single analysis or transformation and then writes the results out to a file. This is inefficient but flexible. SUIF files always use the same output format so that passes can be reordered simply by running the programs in a different order. New passes can be freely inserted at any point in a compilation. This approach also simplifies the process of making code modifications. If the modifications are performed as the last step in a pass, only the actual SUIF code needs to be changed. Other data structures associated with the code (e.g. flow graphs) are simply reconstructed by the next pass that needs them. Again, this is inefficient, but it eliminates most of the tedious bookkeeping associated with making code modifications in other compilers.

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 The SUIF Compiler Toolkit

The compiler toolkit consists of C and Fortran front ends, a loop-level parallelism and locality optimizer, an optimizing MIPS back end, a set of compiler development tools, and support for instructional use.

3.1 Front Ends

Our ANSI C front end is based on Fraser and Hansen's `lcc` [3], which has been modified to generate SUIF files. We do not have a Fortran front end that directly translates Fortran into SUIF. Instead, we use AT&T's `f2c` [4] to translate Fortran 77 into C, followed by our C front end to convert the C programs into SUIF. We have modified `f2c` to capture some of the Fortran-specific information so that we can pass it down to the SUIF compiler. Nonetheless, the conversion to C occasionally obscures the high-level program semantics originally available in Fortran programs. 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.

3.2 A Loop-Level Parallelism and Locality Optimizer

To achieve good performance on modern architectures, programs must make effective use of the computer's memory hierarchy as well as its ability to perform operations in parallel. A key element of the SUIF compiler toolkit is thus a library and driver for applying loop-level parallelism and locality optimizations.

The SUIF parallelizer translates sequential programs into parallel code for shared address space machines. The compiler generates a single-program, multiple-data (SPMD) program that contains calls to a portable run-time library. We then use our SUIF-to-C translator to convert SUIF into ANSI C. This translation enables us to apply our high-level transformations and parallelization techniques on machines for which we cannot directly generate object code. We currently have versions of the run-time library for SGI machines, the Stanford DASH multiprocessor [6], and the Kendall Square Research KSR1. We have also implemented a uniprocessor version of the library used for debugging and testing.

The SUIF parallelizer 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.

Loops containing reductions can be parallelized with simple synchronization. We have an analyzer that recognizes sum, product, minimum, and maximum reductions. It is sophisticated enough to recognize reductions that span multiple, arbitrarily nested loops. Our algorithm tries to find the largest region of code whose accesses to a scalar variable or sections of an array are all commutative “read-modify-write” operations. This definition even allows us to find a reduction that accumulates to a section of an array indirectly through an index array.

The parallelizer performs data dependence analysis using the SUIF dependence library. Traditional data dependence analysis determines if there is an overlap between pairs of array accesses whose indices and loop bounds are affine functions of loop indices. Our dependence analyzer is based on the algorithm described in Maydan *et al.*’s paper [8]. It consists of a series of fast exact tests, each applicable to a limited domain. Its last test is a Fourier-Motzkin elimination algorithm that has been extended to solve for integer solutions. The algorithm also uses memoization, the technique of remembering previous test results, to capitalize on the fact that many dependence tests performed in one compilation are identical. The technique has been shown to efficiently generate exact answers to all the data dependence tests invoked on the Perfect Club benchmarks. We have also extended our dependence analyzer to handle some simple nonlinear array accesses.

The loop transformer is based on Wolf and Lam’s algorithms [13, 14]. The parallelizer optimizes the code for coarse-grain parallelism via unimodular loop transformations (loop interchange, skewing, and reversal) and blocking (or tiling). It can also optimize for cache locality on uniprocessors and multiprocessors. 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.

3.3 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. Examples of such projects that we have implemented with SUIF include a software prefetching tool [9] and an instruction scheduler for superscalar processors [10]. To support such work, the initial release of the SUIF toolkit includes a set of conventional data-flow optimizations and a MIPS code generator.

The data-flow optimizer is built using Sharlit—a data-flow optimizer generator that automatically takes a data-flow description and translates it into efficient optimization code [11]. The optimizations implemented using Sharlit include constant propagation, partial redundancy elimination, strength reduction, and register allocation.

3.4 Compiler Development Tools

The compiler toolkit includes many facilities to aid in the compiler development process. We have identified common functions needed by many of the compiler passes and made them into a set of libraries, some of which are described below. Several debugging tools are also available. A library of functions to check for correct types and consistent data structures helps to identify errors as soon as they occur instead of waiting for them to break subsequent passes. More subtle errors typically require examination of the SUIF code. SUIF binary files can be viewed in several different formats: as text directly corresponding to the SUIF representation, as formatted PostScript showing the tree structures, or as high-level C code.

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.

3.5 A Simplified SUIF for Instructional Use

Students just learning about compiler optimizations would be overwhelmed by the task of implementing them in the context of a complete compiler, so we have developed the Simple-SUIF library to provide a simplified interface to the SUIF system. 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.

4 Conclusion

The SUIF system is capable of compiling standard benchmark programs. We have run our compiler on the Multiflow test suite, the Perfect Club benchmarks, and the SPEC92 benchmarks. Using the MIPS back end without optimization[2], we are able to compile and validate all of these programs. The parallelizer has been successfully tested on the Perfect benchmarks, the Fortran floating-point applications in SPEC92, and the NAS parallel benchmarks.

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.

How to get the SUIF software: The SUIF system is now freely available by anonymous `ftp` from `suif.Stanford.EDU`. Additional information about SUIF can be found on the World-Wide Web at `http://suif.Stanford.EDU`. The SUIF system is implemented in C^{++} and contains over 200,000 lines of code. The release comes with full documentation on the design of the kernel and the libraries and with usage information for all the compiler passes in the toolkit.

5 Acknowledgments

We would like to acknowledge the previous members of the SUIF research team for their contributions to this release: Dror Maydan for his data dependence analysis, Todd Mowry for shaking out bugs in the compiler, Karen Pieper for her help on the original SUIF system design, Mike Smith for his MIPS code generator, Todd Smith for his work on the translators between C and SUIF, and Michael Wolf for building the initial system as well as the loop transformation library. We would also like to thank the other current members of the SUIF compiler team, Amy Lim, Brian Murphy, and Patrick Sathyanathan. They helped with this release and are developing software to be incorporated in the future. We also want to thank John Ruttenberg for letting us use the Multiflow test suite.

The SUIF compiler project has been supported in part by DARPA contracts N00014-87-K-0828, N00039-91-C-0138, and DABT63-91-K-0003. It has also been supported by an NSF Young Investigator Award, NSF CISE Postdoctoral Fellowships, and student fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, Intel, and NSF.

[2]Our scalar optimizer is known to fail under some circumstances; it will eventually be replaced.

References

[1] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[2] J. M. Anderson and M. S. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[3] C. W. Fraser and D. R. Hanson. A Retargetable Compiler for ANSI C. *SIGPLAN Notices*, 26(10), Oct. 1991.

[4] D. Gay, S. Feldman, M. Maimone, and N. Schryer. f2c. Available via netlib@research.att.com.

[5] M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. FIAT: A Framework for Interprocedural Analysis and Transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Aug. 1993.

[6] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. L. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[7] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array Data Flow Analysis and its Use in Array Privatization. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1993.

[8] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[9] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[10] M. D. Smith, M. A. Horowitz, and M. S. Lam. Efficient Superscalar Performance Through Boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, Oct. 1992.

[11] S. W. K. Tjiang and J. L. Hennessy. Sharlit—A Tool for Building Optimizers. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[12] S. W. K. Tjiang, M. E. Wolf, M. S. Lam, K. Pieper, and J. L. Hennessy. Integrating Scalar Optimizations and Parallelization. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 137–151, Aug. 1991.

[13] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[14] M. E. Wolf and M. S. Lam. A Loop Transformation Theory and An Algorithm to Maximize Parallelism. *IEEE Transactions on Parallel and Distributed Systems*, pages 452–471, Oct. 1991.