An Overview of the SUIF Compiler System

Robert Wilson, Robert French, Christopher Wilson,
Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih-Wei Liao,
Chau-Wen Tseng, Mary Hall, Monica Lam, and John Hennessy

Computer Systems Lab
Stanford University, CA 94305

1 Introduction

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. However, such a compiler infrastructure is crucial to the advancement of compiler research, since new technology must be evaluated in combination with all other standard optimizations to improve the speed of large programs. There is a great need for the compiler research community to share their investments in compiler infrastructure development.

At Stanford University, we have been developing the SUIF (Stanford University Intermediate Format) compiler system as a platform for research on compiler techniques for high-performance machines. Our group has successfully used this system for compiler research on topics including scalar data flow optimizations, array data dependence analysis, loop transformations for both locality and parallelism, software prefetching, and instruction scheduling. Ongoing research projects using SUIF include global data and computation distribution for both shared and distributed address space machines, array privatization, interprocedural parallelization, pointer analysis, and optimizing Verilog simulations. SUIF has also been used for courses on compiler optimizations at Stanford.

Encouraged by our positive experiences with the SUIF system, we have undertaken a considerable effort to make the system usable by other research groups. 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`. This release of SUIF is intended for research and personal use only; redistribution and commercial uses are forbidden without prior written permission. The system is provided with no warranty. While we are interested in feedback, we cannot promise to respond to all bug reports.

SUIF is a research compiler. It is designed to facilitate experimentation and development of new compiler algorithms, ranging from high-level transformations to conventional data flow optimizations. The current release is a major revision from an earlier version [9]. In the interest of making the system available as soon as possible, this initial release is a minimal system that we believe is useful to some compiler researchers. Not all the passes we have developed in the previous system have been updated. There are also a lot of experimental passes that are not yet ready for release. We plan to add to the release as time progresses.

The initial system includes a parallelizer that can automatically find parallel loops in Fortran and C programs and generate parallelized C code. C is used here as a portable machine language—our parallelized C code can be compiled by standard C compilers for different machines. Besides generating C, our system can also generate optimized MIPS code. The system provides other features necessary for parallelization: data dependence analysis, reduction recognition, a set of conventional symbolic analyses to improve the detection of parallelism, and unimodular transformations to increase parallelism and locality. Conventional data flow optimizations such as partial redundancy elimination and register allocation are also included. While the system is not as robust as commercial compilers, it is capable of compiling standard benchmark suites.

2 Our Engineering Approach

The task of building a compiler infrastructure that is suitable for everyone is not within the means of our research group. Thus, our goal is only to construct a base system that can support collaborative research and development efforts. To this end, 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 is designed to:

- make all program information necessary for scalar and parallel compiler optimizations easily available;
- foster code reuse, sharing, and modularity; and
- support experimentation and system prototyping.

This design makes it easy to enhance, replace, and add compiler optimizations to the toolkit.

The design of our compiler has also made it practical for us to develop a usable system. While it is important that the toolkit constitutes a fully functional compiler, it need not be a complete suite of perfectly implemented optimizations before the system is usable as a research tool. We have thus focused our energy on the design, development, and documentation of this kernel, the foundation of the infrastructure.

Compiler researchers wishing to develop a new optimization in SUIF need to learn only the design of the kernel. They can assemble their experimental compiler by combining suitable passes from the toolkit with their own experimental passes. They may wish to share their code once their research is complete. Our goal is to establish a standard interface between compiler passes so that other researchers can collaborate in developing a common research infrastructure. We view our compiler system as a bootstrapping system. The kernel will continue to grow as experimental ideas mature. The various compilation passes in the toolkit, however, may one day all be replaced with passes embodying newer compilation techniques and better engineering designs.

3 The Kernel Design

The kernel performs three major functions:

- *It defines the intermediate representation of programs.* The program representation is designed to support both high-level program restructuring transformations as well as low-level analyses and optimizations.
- *It supports a set of program manipulation primitives.* Centralizing the routines that manipulate programs directly makes it easier to implement changes in the program representation.
- *It structures the interface between different compiler passes.* Compilation passes are implemented as separate programs that communicate via files. Different passes can communicate by annotating the program representation. The system supports experimentation by allowing communication of data structures unknown to the kernel.

In the following, we expand on each of these aspects of the kernel. More information is available in the SUIF Library Reference Manual[1].

3.1 Program Intermediate Format

Traditional compilers for uniprocessors generally use program representations that are too low-level for parallelization. For example, it is difficult to extract data dependence information once the data accesses are expanded into simple arithmetic operations. Many parallelizing compilers are built as source-to-source translators and the analyses and optimizations work directly at the abstract syntax tree level. 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, it would be difficult to extend a Fortran 77 compiler to accept Fortran 90 programs. 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, our representation includes three high-level constructs: loops, conditional statements, and array access operations. The loop and conditional representations are similar to abstract syntax trees but are language-independent. These constructs capture all the high-level information useful for parallelization. This approach reduces all the different ways of expressing the same information to a canonical form, thus simplifying the design of the analyzers and optimizers. Results from the analysis of the high-level constructs can easily be carried down to the low-level representation. For example, results from data dependence analysis can be passed down for instruction scheduling.

Our program representation also includes detailed symbol and type information. The information is complete enough to translate SUIF back to legal and high-level C code. The system also keeps sufficient type information in the program representation to enable full interprocedural analysis.

3.2 An Object-Oriented Implementation

The SUIF kernel provides an object-oriented implementation of the SUIF intermediate format. The data structures in the internal representation are organized in a hierarchy. A program consists of a set of files, each of which contains a number of procedures. High-level compiler passes typically

[1] http://suif.stanford.edu/suif/docs/suif_toc.html

choose to represent a procedure as a list of language-independent abstract syntax trees (ASTs). The ASTs include block-structured constructs such as loops and if-statements. At the leaves of the ASTs are expression trees, composed of SUIF instructions. Most instructions perform simple RISC-like operations, but in high-level SUIF the array accesses are not reduced to arithmetic address calculations. Unstructured or irregular control flow is simply represented by lists of instructions that include branch and jump operations. Low-level compiler passes typically expand the ASTs, array accesses, and expression trees into simple lists of instructions.

Provided along with these data structures are various functions to manipulate them. They include functions to read and write SUIF files, traverse the program representation, and create new instructions and symbols. Other functions are available for common operations such as duplicating regions of code.

3.3 Interface Between Compilation Passes

The SUIF system 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.

Different compiler passes interact with one another either by updating the SUIF representation directly or by adding annotations to the SUIF program. Each kind of annotation is defined 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 module with another module that generates the same annotations. The annotation facility encourages experimentation with new abstractions. New definitions of annotations can easily be added as new program abstractions emerge. Once the functions and design of the abstractions are understood, we can then formally introduce them into the compiler kernel.

4 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. Appendix A illustrates how the elements of the toolkit can be combined in a compiler.

4.1 Front Ends

Our C front end is based on Fraser and Hansen's `lcc` front end [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 `f2c` [4] to translate Fortran 77 into C and then use 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 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 unfortunately we have not found time to work on it. In the meantime the current solution works for most benchmarks and is sufficient for research purposes.

4.2 A Parallelizer and Locality Optimizer

Our parallelizer uses both scalar data flow analysis and array data dependence analysis in optimizing for locality and finding parallelism. Our data flow analyzer performs constant propagation and forward propagation, detects multi-dimensional induction variables, and privatizes scalar variables. Except for scalar privatization, the algorithms used in these passes are not very sophisticated. Our long term plan is to reimplement the scalar optimizations and replace our current analyzers.

We have an analyzer that recognizes sum, product, minimum, and maximum reductions. The analyzer 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 allows us to even find reductions that accumulate to a section of an array indirectly through index arrays.

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[2] is based on the algorithm described in Maydan et al.'s paper [6]. 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.

Our loop transformer is based on Wolf and Lam's algorithms [11, 12]. 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 not just on loops with distance vectors but also those with 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 the loop transformation library[3]. Other researchers can use this library to implement different loop transformation policies.

Our parallel code generator[4] changes the sequential code to exploit the coarsest granularity of parallelism available and inserts calls to our run-time system. We have a C code generator which translates SUIF back to C. This enables us to apply our high-level transformations and parallelization techniques on machines for which we cannot directly generate object code. Included

[2]http://suif.stanford.edu/suif/docs/dependence_toc.html
[3]http://suif.stanford.edu/suif/docs/transform_toc.html
[4]http://suif.stanford.edu/suif/docs/parguide_toc.html

in this release is a run-time system that supports parallel execution on SGI machines and the Stanford DASH multiprocessor [5].

4.3 An Optimizing MIPS Back End

While our focus is not on traditional compiler optimizations for sequential machines, some compiler research projects need access to an optimizing back end. Examples of such projects that we have implemented in SUIF include a software prefetcher [7] and an instruction scheduler for superscalar machines [8]. (We plan to release the software prefetcher and instruction scheduler in the future.)

The initial release includes a set of conventional data flow optimizations and a MIPS code generator. However, this back end still uses the previous generation of the SUIF program representation. We can use this back end by first running a translator that converts programs in our current SUIF representation to the older representation.

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 data flow optimization code [10]. The optimizations implemented using Sharlit include constant propagation, partial redundancy elimination, strength reduction, and register allocation.

4.4 Compiler Development Tools

The compiler toolkit includes several tools to facilitate the compiler development process. For example, we have compiler passes that translate a SUIF binary file into either a textual form or a pretty PostScript output. We have also identified some common functions needed by many of the compiler passes and have made them into a set of libraries. Some examples of using these tools to write new SUIF passes are provided in the SUIF cookbook[5].

The mathematics library[6] provides a set of common mathematical functions that are used by the SUIF compiler. The data dependence analyzer and code generator rely heavily on operations on integer matrices and linear inequalities. Included in the library is a Fourier-Motzkin elimination algorithm for solving systems of real 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. We have also developed an interactive interface to this mathematics library, which we call LIC[7] (Linear Inequality Calculator). This tool allows a compiler developer to easily test out new algorithms on examples. This system has also been used in a Stanford compiler course to aid students in learning about data dependence analysis and code generation.

High-level program transformations often need to generate a lot of new code. It is not uncommon to have to update a few hundred data structures to generate a small number of lines of C code. The builder library[8] allows the user to generate C statements with ease similar to that of writing the C statements directly. The builder also insulates the user from future changes to the internal program representation.

The checksuif library is a debugging tool that checks the consistency of the SUIF data structures. This is especially useful for developing sophisticated optimizers that make massive changes to the

[5] http://suif.stanford.edu/suif/docs/cookbook_toc.html
[6] http://suif.stanford.edu/suif/docs/suifmath_toc.html
[7] http://suif.stanford.edu/suif/docs/lic_toc.html
[8] http://suif.stanford.edu/suif/docs/builder_toc.html

original code. Another library provides a collection of miscellaneous useful functions.

4.5 A Simplified SUIF for Instructional Use

We have also developed a Simple-SUIF library[9] to provide a simplified interface to the SUIF system for instructional use. This interface hides most of the complicated details of SUIF so that students only need to deal with the basic concepts. This system has been used in a compiler optimization course at Stanford. Students were able to use SUIF to develop their own data flow optimizations in the context of a fully functional C compiler within a quarter-based course.

5 Current Research Topics

SUIF is a growing system. We are continually developing and experimenting with new compiler techniques. We will release the software we are developing as it matures.

One of the major thrusts in our compiler research is to develop program optimizations to improve both parallelism and locality together. We are currently developing an integrated program transformer that can apply unimodular transforms, loop distribution, loop fusion, and loop reindexing to sequences of arbitrarily nested loops to increase parallelism and locality. We are experimenting with an algorithm that can automatically decompose data and computation across multiple processors to minimize communication while preserving parallelism [2]. We are also developing algorithms to restructure arrays to enhance the performance of the memory hierarchy in a shared address space machine and algorithms to generate optimized communication code on a distributed address space machine [1].

Interprocedural analysis is another current research focus of ours. We have prototype implementations of a scalar data flow analyzer that can find loop induction and loop invariant variables across procedures and also propagate constants and linear relationships between variables across procedural boundaries. We are also working on an array analyzer that can find interprocedural data dependence and data flow relationships among individual array element accesses. We are currently experimenting with interprocedural array privatization and interprocedural reduction recognition. We are also researching fast pointer analysis techniques.

Finally, a third and more recent research goal is to apply our compiler technology to speed up simulations of hardware designs written in Verilog. Our machine targets include superscalar processors and small-scale multiprocessors.

6 Status of the SUIF System

We have found that a compiler infrastructure is only as strong as the set of compiler passes that have been implemented on top of it. Distinctly new types of analysis often require program information that was not captured by the original program representation. For example, our earlier design did not capture enough information to enable a pass that converts SUIF back to C. Our first prototype implementation of an interprocedural analyzer also uncovered serious deficiencies in our representation of Fortran type information. All these different research project experiences have fed back to the design and improvement of the SUIF kernel. So far, six Ph.D. theses have been

[9] http://suif.stanford.edu/suif/docs/simple_toc.html

completed using SUIF, and preliminary implementations exist for three more thesis projects. We believe that the system is general enough to support a wide variety of research topics.

The strength of SUIF lies not just in the functionality offered in the initial release but in its engineering to support new research and development. Over the course of the last five years, SUIF's support of code development has improved significantly. As our understanding of high-level transformations grows, we discover general abstractions that are useful to various program analyses and optimizations. These abstractions lead to better program representations and to the development of generally useful libraries. One such example is the builder library which provides a general facility for generating new code. We have been pleasantly surprised to find how quickly we can develop substantial compiler passes in this system.

The system has also been demonstrated to support collaborative research. About twenty researchers have worked on the SUIF system in the last five years, and roughly eight researchers worked on the system at any given time.

Our compiler system is implemented in C++. It consists of about 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.

SUIF is not a production compiler. As a research vehicle, SUIF is designed to support modularity and experimentation at the cost of compilation speed. The design choice of having all compilation passes communicate via files is a clear example of such a trade-off. With research as our primary mission and the resources of only about ten people, the SUIF system is not as robust as commercial compilers. Nonetheless, our compiler has been known to compile large programs correctly and is robust enough to support experimental research.

At the time of the current release, we have run our compiler on the Multiflow C test suite, the Livermore kernels, the PERFECT Club benchmarks, and the SPEC92 benchmarks. The PERFECT benchmarks are fourteen programs, ranging in size from 500 to 18,000 lines of Fortran, and have a total of 65,000 lines of code. There are fourteen floating-point applications in the SPEC92 benchmarks, ranging in size from 200 to 18,000 lines of code, with a total of 57,000 lines. In addition, there are six integer applications in the SPEC92 benchmarks. They contain 118,000 lines of C code, including 84,000 in `gcc`.

Using the MIPS back end without optimization, we are able to compile and validate all of the PERFECT and SPEC92 benchmarks, and all of the Livermore kernels. Our scalar optimizer currently has several known bugs. With optimizations turned on, the system compiles two thirds of the programs correctly. (Details of the compiler status are included in the release notes.) Preliminary evaluation indicates that our optimized code usually runs within 70–90% of the performance of commercially available compilers on MIPS architectures.

The parallelizer has been tested on all the PERFECT Club benchmarks, the Fortran floating-point applications in SPEC92, and the NAS parallel benchmarks. We used our compiler to generate parallelized C code which was then compiled by the native compiler. The resulting parallel code for all these benchmarks validated. However, we have not yet tuned the compiler or the run-time system to improve its parallel execution performance. We have also compared our parallelizer with KAP, a commercially available compiler. KAP has a number of analyses and optimizations that the current release of SUIF does not perform. To test the effectiveness of our implementation, we ran KAP with only those analyses and optimizations included in SUIF. Our parallelizer finds roughly 95% of the parallel loops found by the KAP compiler and can occasionally detect parallel loops not found by KAP. Details of the parallelizer's performance can be found in the SUIF Parallelizing Compiler Guide[10]. Our results suggest that our basic parallelizer system provides a solid platform for experimentation with advanced compiler techniques.

7 Concluding Remarks

The strength of SUIF lies not just in the functionality offered in the initial release but in its engineering to support collaborative research in both sequential and parallelizing analyses and optimizations, both within and across procedures. The best demonstration that an infrastructure supports a wide range of compiler optimizations is to build them. We have used SUIF to develop many different compiler techniques, and many prototype implementations are in progress. The SUIF system design has benefited from our research program and has gone through several major revisions and improvements. There is still a long way to go in developing the compiler toolkit. We welcome contributions by other research groups to improve upon this compiler infrastructure.

8 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 next release. 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, Postdoctoral Fellowships from NSF, and student fellowships from AT&T Bell Laboratories, DEC Western Research Laboratory, Intel, and NSF.

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 David R. Hanson. A retargetable compiler for ANSI c. *SIGPLAN Notices*, 26(10), October 1991.

[10] http://suif.stanford.edu/suif/docs/parguide_toc.html

[4] D. Gay, Stu Feldman, Mark Maimone, and Norm Schryer. f2c. Available via netlib@research.att.com.

[5] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[6] 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.

[7] Todd C. Mowry, Monica S. Lam, and Anoop 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, October 1992.

[8] Michael D. Smith, Mark A. Horowitz, and Monica 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, October 1992.

[9] S. W. K. Tjiang, M. E. Wolf, M. S. Lam, K. Pieper, and J. L. Hennessy. *Integrating Scalar Optimizations and Parallelization*, pages 137–151. Springer-Verlag, August 1991.

[10] Steven W. K. Tjiang and John 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.

[11] 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.

[12] 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, October 1991. also available as technical report CSL-TR-91-457.

A Features of the SUIF Toolkit

The SUIF toolkit includes a variety of passes. Figure 1 shows how these passes can be combined to produce a complete compiler. This is how roughly how the SUIF compiler driver runs the passes.

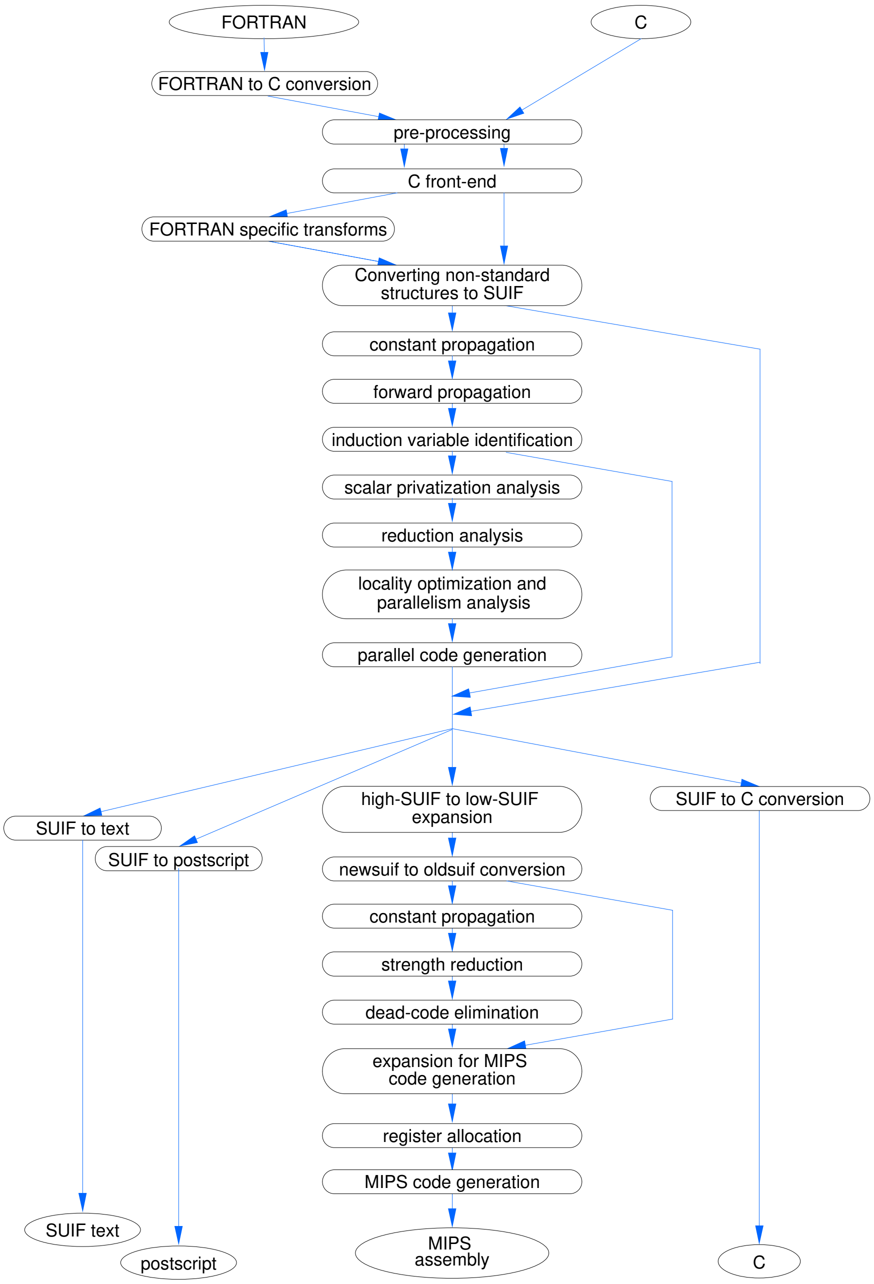

Figure 1: Sample Usage of the SUIF Toolkit