Department of Computer Science

Graphical User Interface for Compiler Optimizations with Simple-SUIF

Brian Harvey
Department of Computer Science
University of California
Riverside, CA 92521
`briancs.ucr.edu`

Gary Tyson
Department of Computer Science
University of California
Riverside, CA 92521
`tysoncs.ucr.edu`

UCR-CS-96-5

Technical Report

**COLLEGE OF ENGINEERING
UNIVERSITY OF CALIFORNIA
RIVERSIDE**

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Graphical User Interface for Compiler Optimizations with Simple-SUIF

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in
Computer Science

by

Brian Keith Harvey

December, 1996

Thesis Committee:

Professor Gary Tyson, Chairperson

Professor Thomas Payne

Professor Frank Vahid

ABSTRACT OF THE THESIS

Graphical User Interface for Compiler Optimizations with Simple-SUIF

by

Brian Keith Harvey

Master of Science, Graduate Program in Computer Science

University of California, Riverside, December, 1996

Professor Gary Tyson, Chairperson

Very few tools exist which support the process of studying back end optimizations. Currently, researchers examining different back end optimizations must rely on general debugging tools or design their own tools to help in the generation of optimizing functions. A tool designed to give information specifically tailored for optimization designers is necessary. Such a tool would allow researchers to view the results of a newly designed optimization or analyze how the intermediate code of a program changes depending on the order in which transformations are applied. More efficient development of optimization routines should be the result of using this tool.

This thesis presents the Visual Simple-SUIF Compiler (VSSC) package, designed to accomplish these tasks by providing an interactive framework that facilitates the development of code transformations in the back end component of a compiler. Code transformation algorithms are viewed by *stepping* through the transformation while actual changes to the intermediate code are performed visually. Transformations can be applied to the intermediate code in any order and can

be *undone*. Capabilities such as these aid in the educational process of learning optimization algorithms and in the testing of newly developed optimizations.

A tool of this nature requires several key components that integrate well with each other to form a single framework. These components include: an easily extendible compiler with a simple, yet functional, intermediate format, a graphical user interface toolkit to implement the graphical "interactive" component of such a tool, and a subsystem for drawing graphs which are common data structures used in code transformations.

VSSC incorporates a graphical user interface (Tcl/Tk), the Simple-SUIF compiler framework, and the `DOT` graph drawing tool to visually illustrate data flow analysis and code transformations. VSSC has many advantages that make it useful to both optimization researchers and students alike.

The goal of this thesis to describe the design and implementation of a framework which integrates these components as well as outlining the benefits the use of a tool based on this framework provides.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Compilers play an important part in the field of Computer Science; they take a program that we have written and translate it to a form which can be executed on a computer. This can be done by reading a program written in a high-level language, called the source language, and translating it into an equivalent program in another language called the target language. A variety of compilers exist today because there are numerous source languages and many different target languages. Target languages range from the machine language for one of numerous different architectures to other high-level programming languages.

It is a common misconception that the process of compiling source code into an executable is performed by a single program. In actuality, this process is performed by an entire set of programs, each with its specific task, that together form a language-processing system [1]. This system typically includes: a *preprocessor* which performs macro expansion, header file inclusion, and perhaps language extensions, a *compiler* which takes this modified source code and translates it into equivalent assembly code, an *assembler* which converts assembly code generated by the compiler into relocatable machine code, and a *loader/linker* which takes different modules of machine code and libraries and combines them into a single executable, altering reference addresses as needed. This language-processing system and its components are shown in Figure 1.1. This figure shows how the original source code transforms into the final executable program.

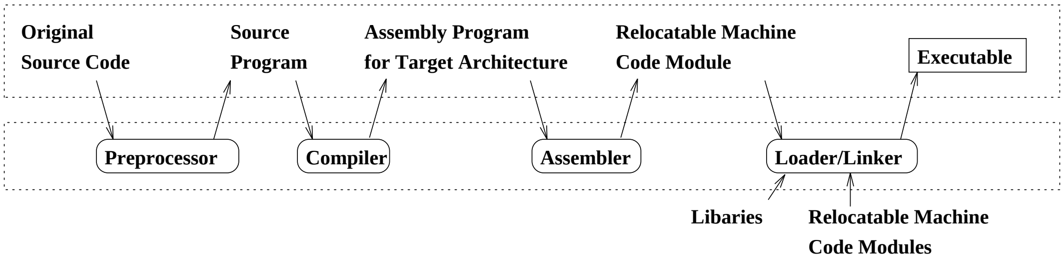

Figure 1.1: Language-processing system

The functionality of the compiler module in a language-processing system can be separated into two main components [1]. The *front end* takes source code, performs lexical, syntactic, and semantic analysis on it and produces an intermediate representation of that code. The *back end* takes the intermediate representation, optimizes it, and generates assembly code for the target machine. As shown in Figure 1.2, the intermediate representation acts as the *glue* that connects the front and back ends together to produce a compiler.

1.1 Front End

The front end takes source code and converts it to a format that is an intermediate representation of that source code. This process includes checking the syntax of the source code and checking the semantics of the source code. There exist several well-known tools to help compiler writers with the front end. These tools, which

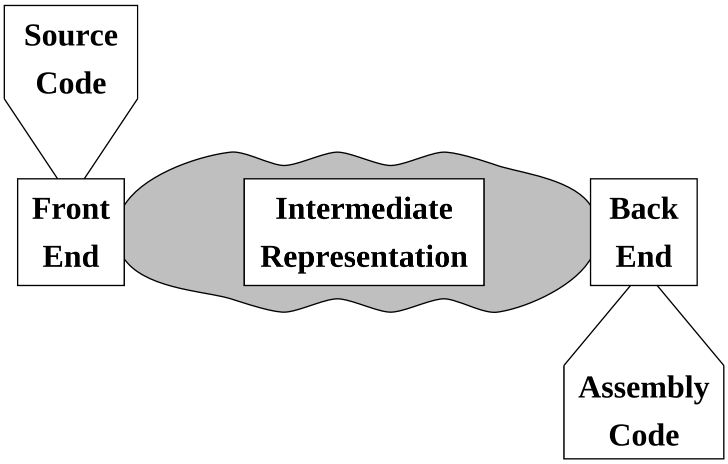

Figure 1.2: Simplified model of compiler

include lex [23], yacc [20], and ANTLR [29], use a specification of the source language to generate a translator from the source language to an intermediate specification. This is often accomplished by using combinations of these tools targeting each to a portion of the translation. For example, lex generates a lexical analyzer which translates the source tokens to a token stream; similarly, yacc generates a syntactic parser which translates the token stream into a parse tree representation of the original source program. The theory behind the translation process performed by the front end is fairly mature and so are the tools which create these translators. Most of the recent front end research concentrates on the design and implementation of new language features.

1.2 Intermediate Representation

The intermediate representation separates the front end, which deals with source language issues, from the back end, which deals with target issues. There are several properties necessary to make an intermediate representation useful. First of all, it should be easy to generate from a variety of front ends. This allows it to remain useful as languages evolve (e.g. C$\rightarrow$C++). Secondly, the intermediate representation should be easy to manipulate during code transformations. This simplifies the coding of the various analysis and transformations and allows changes to be made to the intermediate representation without difficulty. Finally, it must be machine-independent. This means that the intermediate code knows nothing about the target architecture. For example, it does not care about the number of registers the target provides and instead assumes an unlimited number of registers. Because it is machine-independent, intermediate code usually looks like assembly code for a virtual machine. This machine-independent feature means that the intermediate code can be retargeted to many different architectures simply by using a different back end and the compiler can use a machine-independent code optimizer in the back end.

The importance of a good intermediate representation is prevalent even in today's technologies. Java [18], a popular Internet programming language, translates Java source code into a bytecode format which acts as an assembly language for a virtual machine. This bytecode is the intermediate format between the source code and a running program. When a Java program is "executed", the bytecode is simply interpreted[1]. Java bytecode can be run, without modification, on any operating system that the Java interpreter has been ported to.

[1] Recent advancements have produced Java Just-In-Time compilers, which use the Java bytecode as an intermediate representation for a completely separate compilation.

1.3 Back End

Two tasks of the back end of an optimizing compiler are to modify the intermediate code to improve overall code performance by performing various code transformations and optimizations and to translate the intermediate code to assembly code for the target architecture.

Most recent research in compilers has concentrated around the optimizer component of the compiler. The optimizer determines what the final assembly code will look like and how fast it will run. The task performed by the optimizer must be separated into transformations or *phases* in order to make this enormous task manageable. The function of the optimizer is complicated by the fact that transformations cannot always be applied in any order. Additionally, these transformations normally cannot be combined because each transformation usually performs a specific task which requires data flow information useful for only that transformation.

The need to separate the optimization phases leads to some difficulties in ordering the transformations. Given a set transformations to perform, the following situation can occur: Once a certain optimization is done, a subsequent optimization may be unable to make any useful changes because of the changes made in the first transformation; whereas, if this second optimization had been executed before the other one, its transformations would have been more beneficial. Unfortunately, it is very difficult for a compiler back end to determine in what *order* the optimizations should be made.

This problem, called the *phase-ordering* problem, has been studied for many years. One example of this problem deals with register allocation and instruction scheduling. If register allocation is performed before scheduling, it may introduce artificial data precedence, keeping the instruction scheduler from generating the best schedule. However, performing register allocation after instruction scheduling may result in the need for more registers. The phase-ordering problem can be difficult to avoid. Research in this area has proposed several solutions for specific transformations in which the phases are combined. A framework for describing optimizations and an example framework in which constant propagation, value numbering and unreachable-code elimination are combined is presented in [7]. Combining these phases allows for more information about the program to be discovered and thus more opportunities for optimization. In [25], register allocation and instruction scheduling are combined. A heuristic algorithm is used in which weights are used for controlling register pressure and instruction parallelism. Finally, a compiler in which the code generation phase and a machine-directed peephole optimizer are tightly integrated is described in [13] and [14]. These two components can be combined because they are both simple pattern matchers. A peephole optimizer replaces patterns of code with more efficient code while a code generator matches patterns of intermediate code and replaces them with assembly code. This compiler uses a single rule-based pattern matching system which makes the compiler simple, fast, and retargetable.

1.3.1 Constructing Code Optimizers

Currently, few tools exist to help construct code optimizers. This is slowly changing as the importance of optimization grows. One such tool to appear is **Sharlit** [39]. Built to work with the SUIF compiler system, **Sharlit** helps in constructing data flow analyzers and the transformations that use data flow analysis information. Compiler writers are able to construct global analyzes and optimizations with the following modular components: flow graphs, flow values (values that flow through the graph), flow functions that represent the effect of flow graph nodes and paths on flow values, action routines that are used to perform program optimizations based on the previous data flow analysis, and path simplification rules that show how to combine flow functions into other flow functions.

Like the front end of a compiler, the code generation stage of the back end is not as actively studied because the process is relatively simple: convert from one representation to another. Some tools exist to help compiler writers with this functionality of the back end. These tools, such as `iburg` [11], `burg` [12], and `Twig` [38], act as code generator *generators*. Each of these tools reads in a specification and generates C code to perform the code generation based on the specification. The specification usually specifies the cost of operands and instruction in the intermediate format. These tools use tree pattern matching and dynamic programming to produce a code generator. The main difference between the various code generator generators mentioned is how they implement their tree pattern matching and whether the dynamic programming is done when the tool is used or if it is embedded in the generated code generator.

1.4 Design Tools for Developing Optimizers

There are few tools which support the process of studying back end optimizations. Currently, researchers examining different back end optimizations must rely on general debugging tools or design their own tools to help in the generation of optimizing functions. A tool designed to give information specifically tailored for optimization designers is necessary. This tool would give researchers the ability to analyze how their new optimization (or existing optimizations) and the intermediate code are affected when performing optimizations in different orders. More efficient development of optimization routines should be the result of using such a tool.

In order to develop such a tool, the intermediate format used in the compiler needs to be well designed and should not change between code transformations; the SUIF (Stanford University Intermediate Format) compiler [42] has such an intermediate format. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. Each pass is implemented as a separate program which reads in the SUIF representation of the source program from a file generated by the front end, performs some code transformation, and then writes out the resulting SUIF representation to a file. Since the SUIF format never changes, these passes can be performed in any order.

SUIF provides the necessary platform for true development of a tool designed to support the development of code transformations. The work described in the remainder of this thesis develops one such tool, the Visual Simple-SUIF Compiler package.

1.5 VSSC

The topic of this thesis is the Visual Simple-SUIF Compiler (VSSC) package which we have developed. This package is designed to provide an interactive framework that allows the development of compiler optimizations. It incorporates a graphical user interface with an underlying Simple-SUIF compiler framework to illustrate data flow analysis and code transformations. VSSC has many advantages that make it useful to both optimization researchers and students alike.

Built on top of the SUIF compiler, VSSC inherits all the strengths of SUIF as well as adding new ones. As in the SUIF compiler, it is easy to add new optimizations to a VSSC compiler and to perform optimizations in any order. The user can select which optimizations to perform through the use of a graphical interface. Transformations can be applied in any order and as many times as desired. This "interactive" ability allows the user to see the effects different code transformations have on the intermediate code after each transformation, as well as viewing the effects of different implementations of a particular code transformation (e.g. different methods of doing dead code elimination). This ability also allows a researcher to see the different results of the intermediate code depending on the order that the transformations are performed. In addition, the researcher can select a transformation to be performed based on the observed characteristics of the current state of the intermediate code. VSSC allows transformations to be *undone*, so the user can perform an optimization, undo it, and then perform another optimization. Another advantage provided by VSSC is that its GUI library allows the optimization designer freedom in controlling what information the GUI displays to the user.

A VSSC compiler runs until the user exits allowing intermediate code to be studied indefinitely between code transformations and viewed at each step during a code transformation. VSSC provides the ability for the optimization writer to display a graph during the transformation. Table 1.1 shows some of the possible types of graphs that can be displayed using VSSC. Examples of these graphs will be demonstrated in Section 5.2.

Showing a graph during the code transformation aids the user in understanding

Type of Graph	**Purpose in a Compiler**
Flow Graph	This type of graph illustrates the flow-of-control information for a procedure. Each node in the graph represents a basic block, a sequence of consecutive statements in which flow of control enters only at the beginning and exits only at the end. There is a directed edge in the graph from block B_i to block B_j if B_j can immediately follow B_i in some execution sequence.
Directed Acyclic Graph	In a DAG, the information of how the value computed by each statement in a basic block is used in subsequent statements of the same block is shown. This information can be used to find common subexpressions (those expressions which are computed more than once can be eliminated). Leaves in this graph are unique identifiers (variable names of constants) and interior nodes are mathematical operators.
Register-Interference Graph	This type of graph is sometimes used when implementing register allocation using graph coloring. The nodes in this graph are symbolic registers and an edge connects two nodes (registers) if one register is live at a point where the other is defined.

Table 1.1: Some of the types of graph that can be constructed and displayed in VSSC

the steps in the code transformation as well as emphasizing the importance of that graph data structure in the code transformation. For example, a flow graph allows the user to view the various possible execution paths between basic blocks in the intermediate code.

Perhaps one of the most important benefits provided by VSSC is the ability to *step* through an optimization. Much in the same way that you can step through code in a code debugger such as GDB [35], the VSSC package allows the optimization writer to configure *steps* within the optimization and when the optimization is performed in the VSSC compiler, the GUI user can *step* through the optimization at his or her own pace. The optimization writer has complete freedom to include whatever actions he or she likes within each step.

Finally, since people tend to learn better by visualization as well as being able to step through an optimization at their own speed, the VSSC package is particularly helpful when used in the academic environment. VSSC can be used as a teaching tool in courses on optimizing compilers. Students taking a course in optimizing compilers can use VSSC in two ways. The first way in which it could be used is that the instructor provides to the students a ready-made VSSC compiler which already implements various optimizations. The students can then use VSSC to augment their learning of an optimization presented by their instructor by *stepping* through it with VSSC on any C code they wish. In this way, VSSC can be a teaching tool for the study of compiler back ends.

Students can also benefit from using VSSC when writing their own optimizations. In a typical optimizing compiler course, students implement such standard optimizations as dead-code elimination, common subexpression elimination, and others described in [1]. Students can use VSSC to facilitate their understanding of what happens during one of these optimizations. VSSC can show them the results of a particular data flow analysis and the effects of a transformation. Students can learn only so much from a textbook and from trying out an optimization on paper. They can gain more insight about the optimization when they can actually see it in action on the screen in front of them. VSSC has been used in the graduate compiler course at the University of California, Riverside. Students implemented basic block detection, various data flow analysis, and register allocation using VSSC.

1.6 Thesis Organization

Chapter 2 provides background information on various retargetable compilers and the intermediate representations they use. It concludes with a description of the compiler used by VSSC. Chapter 3 includes descriptions of work in the area of compiler visualization tools and previous work related to VSSC. Chapter 4 describes the overall design, organization, and implementation of VSSC. Examples of a VSSC compiler performing various optimizations, as well as various types of graphs that can be displayed with VSSC, are shown in Chapter 5. Chapter 6 provides a conclusion along with a discussion of possible future directions for VSSC. Appendix A provides a user's manual for VSSC that describes VSSC's application program interface (API), how to get started using VSSC, and various tips and suggestions on the use of the package.

Chapter 2

Retargetable Compilers

In order to implement the functionality proposed in VSSC, an easily extendible compiler with a robust intermediate format was needed. This chapter discusses several free C compilers including the intermediate format that they use. Their various strengths and weaknesses will be presented as well as an analysis of their suitability for the VSSC framework.

2.1 Requirements for a Retargetable Compiler

A retargetable compiler is one that can support multiple targets by incorporating multiple back ends to generate code for different target architectures. This capability of a compiler is strongly influenced by the intermediate language used by the compiler. In order to make retargeting easy, the intermediate language needs to be machine-independent. If it doesn't rely on the characteristics of the target architecture, then a code generator can be written to generate code from that intermediate code for virtually any architecture.

Academic and research groups tend to use retargetable compilers to allow for greater flexibility in educational and research goals. Because they are retargetable, they usually also run on the various architectures in use at that institution. It is not surprising that retargetable compilers are better organized than their monolithic counterparts. In order to be retargetable, the compiler needs to have a well-defined interface and be modular in design. Most research groups also tend to use *free* retargetable compilers for obvious reasons. In a situation where you want to test out a new compiler feature, it is easier to incorporate it in into an existing compiler for which you have the source code than constructing your own compiler from scratch or purchasing a source code license for a commercial compiler.

In this section we look at three popular retargetable compilers in order to determine which one best supports the VSSC framework. These compilers are: the GNU `gcc` compiler [34] developed by Richard Stallman (and numerous other people), the `lcc` compiler [9] developed by Christopher W. Fraser at AT&T Bell Laboratories and David R. Hanson at Princeton University, and the SUIF compiler [42] developed by Monica Lam at Stanford University. These systems are briefly evaluated below.

2.2 GNU C Compiler (GCC)

The GNU C Compiler (GCC) [34], which can compile C, C++, and Objective-C code, is arguably the world's most popular free compiler. GCC's greatest strength is that fact that GCC has been ported to and has been retargeted to many different operating systems and architectures. This wide-range of use is made possible by GCC's robust intermediate format.

GCC's front end converts the source code into a Lisp-like register transfer language (RTL). The RTL describes each target-dependent instruction in a target-independent algebraic form that defines the semantics of an instruction. After various code transformations are performed on the RTL, GCC's back end takes this RTL and generates code for the target architecture. The specification of the target architecture is based on a machine description that identifies the target code to generate for each possible expression in the RTL.

The widespread use of the GNU Compiler and the large number of contributors to its set of supported optimizations has led to a complex implementation. The GCC design lacks modularity and is somewhat monolithic in design. In addition, the optimization passes made on the RTL have become so dependent on each other over the years that the passes need to be done in a specific order. These limitations make it very difficult to implement the functionality planned for VSSC using the GCC compiler.

2.3 LCC

The `lcc` compiler [9] [10] is an ANSI C compliant retargetable compiler that can generate code for VAX, Motorola 68020, i386, SPARC, and MIPS R3000 architectures. Developed by Christopher W. Fraser at AT&T Bell Laboratories and David R. Hanson at Princeton University, `lcc` is heavily used at both institutions.

`lcc` has many features that make it popular. The simple and compact design of `lcc` makes it one of the smallest and fastest ANSI C compilers available. Probably the most useful feature of `lcc` is the quality of its the documentation. The authors used the **noweb** [31] system to generate a textbook [10] and the source code for `lcc` from a single source. Therefore, the textbook, which describes the implementation of lcc, includes most of the code of lcc along with the explanation of that code and how it contributes to the implementation. When generating the textbook, **noweb** system automatically cross-references all code segments so a reader of the textbook can easily navigate the source code. This is very helpful, since the source code is presented in the textbook in an order that follows the description of the implementation.

The target-independent front end and the target-dependent back end of lcc exist together in a single executable glued together by an efficient interface. This interface consists of only 18 functions and the C code being compiled is represented by a 36-operator dag language [8]. This language is the intermediate format of lcc and represents the source program as it goes from the front end to the back end in the compilation process. The dag language has gone through many changes during the development of lcc. In each change, the authors usually took a more complex operator out of the dag language and added functionality for it in the front end. Consequently, each change made the back end less complex.

lcc has a number of features, which are not normally found in other compilers, that increase its usefulness to the user. Command-line flags can specify that the code generated will check for the dereferencing of a null pointer (a common programming mistake in C), print out function call/return traces, and generate execution profiles. When lcc performs frequency-based profiling, it generates code that keeps track of the number of times each expression is calculated. This allows programmers to try to simplify those expressions that are frequently calculated. The accumulated profiling data can be displayed and analyzed.

lcc, despite its many strengths, has several weaknesses. First of all, it is not an optimizing compiler. While its front end performs some target-independent optimizations such as local common subexpression elimination, constant folding, and other simple transformations, no other optimizations are specified. Other typical compiler systems implement many more global and target-dependent optimizations. Another weakness is the tightly-coupled design between the front and back ends. While this design allows `lcc` to be a small and fast compiler, it can be difficult for a researcher to use `lcc` for a large back-end compilation environment. The design and organization of `lcc` is so highly optimized for compilation speed and code compactness that small changes to one component may greatly affect other components. Other compiler systems, such as SUIF described in the next section, are more modular and robust in their design. This modularity allows a researcher to concentrate only on what they need to, without worrying about how it affects the rest of the compiler system.

2.4 SUIF Compiler System

The SUIF research compiler system [42] [43], developed by a team of researchers under the direction of Monica Lam at Stanford University, is centered around the robust design of its intermediate format called SUIF (Stanford University Intermediate Format). The system has been designed and organized in such a way that it is easy to modify and extend the base system to generate custom compilers. The SUIF team took considerable effort to make the system usable by other research groups. For this reason, many researchers around the world use the SUIF compiler system to evaluate new compiler techniques and perform research on analysis and optimization algorithms.

The SUIF system is organized into two components. The *kernel* of the SUIF compiler defines the central core of the compiler. The design goals of the kernel are [43]:

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

The kernel performs three major tasks: it defines and manages the intermediate format SUIF, it provides a set of routines for manipulating the intermediate format, and it provides an information and communication interface between compiler passes.

2.4.1 SUIF Intermediate Format

The SUIF intermediate format is different from the intermediate formats used by the previously discussed compilers. Those intermediate formats are very low-level while SUIF's intermediate format is a "mixed-level" program representation incorporating both low-level and high-level information. The high-level information includes: loops, conditional statements, and array access operations. The low-level information includes: assembly-like intermediate code, jumps and branches to labels, and symbolic registers. One of the features of the SUIF compiler system is its ability for determining the amount of parallelism in a program[1]. The inclusion of these high-level constructs simplifies the design of analyzers and optimizers. For example, there are many optimizations that deal with loops. However, these

[1] This information can be used to increase program parallelism and locality.

optimizations must detect the loops using data-flow analysis. An optimization in SUIF does not need to perform such analysis, because loop information already exists in the intermediate format. Figure 2.1 shows a sample C program and the SUIF intermediate format of the same example program (using the `printsuif` program[2]).

2.4.2 SUIF Transformation Environent

The second component of the SUIF system is a set of compiler passes that perform various transformations on the intermediate format. Usually, each pass reads in the intermediate code, performs some transformation, analysis, or optimization, and then writes out the intermediate code. Since each pass can exist as a separate executable in the SUIF system, passes can be run in any order in the compilation process. Information can be relayed from pass to pass by annotating components in the intermediate format. To aid in the creation of SUIF compiler passes, the SUIF system contains a robust set of libraries, commonly used data structures, and support routines.

[2]Actually, `printsuif` displays a lot more information about the SUIF intermediate format for this C code, but to save space, only part of the information is shown here for comparison purposes.

```c
int main(int argc, char **argv)
{
   int x, y, z;
   for (x=0; x<100; x++)
   {
      z = 0;
      for (y=0; y<x; y++)
      {
         if (y+z < x)
            z++;
      }
   }
   printf("z= %d\n", z);
}
```

```
  PROC P:.main
  ["line": 2 "example.c"]
 1: mrk
      ["line": 5 "example.c"]
 2: FOR (Index=main.x Test=SLT Cont=L:main.L1 Brk=L:main.L2)
    FOR LB
28:   ldc t:g4 (i.32) 0
    FOR UB
30:   ldc t:g4 (i.32) 100
    FOR STEP
32:   ldc t:g4 (i.32) 1
    FOR LANDING PAD
    FOR BODY
 6:   mrk
        ["line": 7 "example.c"]
 7:   ldc t:g4 (i.32) main.z = 0
 8:   mrk
        ["line": 9 "example.c"]
 9:   FOR (Index=main.y Test=SLT Cont=L:main.L3 Brk=L:main.L4)
      FOR LB
22:     ldc t:g4 (i.32) 0
      FOR UB
        main.x
      FOR STEP
25:     ldc t:g4 (i.32) 1
      FOR LANDING PAD
      FOR BODY
13:     mrk
          ["line": 11 "example.c"]
14:     IF (Jumpto=L:main.L5)
        IF HEADER
15:       bfalse  e1, L:main.L5
16:         e1: sl t:g31 (i.32) e2, main.x
17:           e2: add t:g4 (i.32) main.y, main.z
        IF THEN
18:       mrk
            ["line": 12 "example.c"]
19:       add t:g4 (i.32) main.z = main.z, e1
20:         e1: ldc t:g4 (i.32) 1
        IF ELSE
        IF END
      FOR END
26:   mrk
        ["line": 9 "example.c"]
    FOR END
34: mrk
      ["line": 15 "example.c"]
35: cal t:g4 (i.32) <nullop> = e1(e2, main.z)
36:   e1: ldc t:g39 (p.32) <P:.printf,0>
42:   e2: ldc t:g34 (p.32) <.__tmp_string_0,0>
39: mrk
      ["line": 16 "example.c"]
40: ret  e1
41:   e1: ldc t:g4 (i.32) 0
  PROC END
```

Figure 2.1: C and SUIF intermediate format version of same example program

The SUIF compiler may not be the fastest or the most robust compiler, but the flexibility and extensibility of its design outweighs these possible shortcomings for the study of back-end code transformations. The SUIF compiler is not meant to be a production quality compiler. Instead it is meant to act as a research vehicle, designed to support modularity and experimentation at the cost of the speed of translation.

2.4.3 Simple-SUIF

Since the SUIF compiler is a complete ANSI C compiler, it is a little too complex for use in a college course in compilers. The SUIF Compiler group at Stanford developed a package called **Simple-SUIF** [36] which acts as a *wrapper* for SUIF by providing a simplified interface to the intermediate format generated by the SUIF compiler. When using Simple-SUIF, the SUIF intermediate format remains the same internally, but differs in the way the programmer interacts with the system. This simplified interface allows students to write their own optimizations for a fully-functional ANSI C compiler without learning the high-level constructs required to perform compiler transformations (e.g. interprocedural analysis).

Figure 2.2 shows the Simple-SUIF version of the example code in Figure 2.1. As you can see, the instructions in Simple-SUIF's intermediate format resemble assembly language instructions (*op dst,src1,src2*) or three-address C instructions (*dst = src1 op src2*). Each instruction has an unique opcode associated with it. The instructions are grouped into six different categories called *instruction formats*. Table 2.1 shows all the valid Simple-SUIF instructions. For each instruction, the following information is also shown: the opcode, the Simple-SUIF name, the instruction format, and a short explanation of that instruction.

```
Procedure main:
   ldc  (s.32)  t6 = 0
   cpy  (s.32)  r3 = t6
L9:
   ldc  (s.32)  t7 = 0
   cpy  (s.32)  r5 = t7
   ldc  (s.32)  t8 = 0
   sl   (s.32)  t9 = t8, r3
   bfls         t9, L6
   ldc  (s.32)  t10 = 0
   cpy  (s.32)  r4 = t10
L7:
   add  (s.32)  t11 = r4, r5
   sl   (s.32)  t12 = t11, r3
   bfls         t12, L5
   ldc  (s.32)  t13 = 1
   add  (s.32)  t14 = r5, t13
   cpy  (s.32)  r5 = t14
L5:
L3:
   ldc  (s.32)  t15 = 1
   add  (s.32)  t16 = r4, t15
   cpy  (s.32)  r4 = t16
   sle  (s.32)  t17 = r3, r4
   bfls         t17, L7
L4:
   jmp          __done8
L6:
   ldc  (s.32)  t18 = 0
   cpy  (s.32)  r4 = t18
__done8:
L1:
   ldc  (s.32)  t19 = 1
   add  (s.32)  t20 = r3, t19
   cpy  (s.32)  r3 = t20
   ldc  (s.32)  t21 = 100
   sle  (s.32)  t22 = t21, r3
   bfls         t22, L9
L2:
   ldc  (a.32)  t23 = &printf + 0
   ldc  (a.32)  t24 = &__tmp_string_0 + 0
   call (s.32)  *t23 (t24, r5)
   ldc  (s.32)  t25 = 0
   ret          t25
```

Figure 2.2: SimpleSUIF version of example code used in Figure 2.1

A Simple-SUIF instruction is represented by the **simple_instr** structure. Figure 2.3 shows the **simple_instr** structure and its contents. When writing an optimization pass with Simple-SUIF, the intermediate format is given to the programmer as a linked list of these instructions. The optimization can then manage the items in this linked list. When an optimization is performed, the elements in the linked list are modified and the linked list is returned back to the Simple-SUIF library. This intermediate code is then saved back to a file.

Simple-SUIF Instructions			
Opcode	**Instr. Name**	**Instruction Format**	**Purpose**
No operand instructions			
NOP_OP	nop	BASE_FORM	No nothing at all
One source operand (src1) instructions			
RET_OP	ret	BASE_FORM	Return from a procedure
Two source operand (src1, src2) instructions			
STR_OP	str	BASE_FORM	Store the value in the **src2** register at the address contained in the **src1** register
MCPY_OP	mcpy	BASE_FORM	Memory-to-memory copy
Unary instructions (dst, src1)			
CPY_OP	cpy	BASE_FORM	Copy the **src1** register to the **dst** register
CVT_OP	cvt	BASE_FORM	Convert the **src1** register to the result type and put it in the **dst** register
NEG_OP	neg	BASE_FORM	Negation
NOT_OP	not	BASE_FORM	Bit-wise inversion
LOAD_OP	load	BASE_FORM	Load the value at the address contained in the **src1** register and put it in the **dst** register
Binary instructions (dst, src1, src2)			
ADD_OP	add	BASE_FORM	$\mathbf{dst} = \mathbf{src1} + \mathbf{src2}$
SUB_OP	sub	BASE_FORM	$\mathbf{dst} = \mathbf{src1} - \mathbf{src2}$
MUL_OP	mul	BASE_FORM	$\mathbf{dst} = \mathbf{src1} * \mathbf{src2}$
DIV_OP	div	BASE_FORM	$\mathbf{dst} = \mathbf{src1}/\mathbf{src2}$
REM_OP	rem	BASE_FORM	$\mathbf{dst} = \mathbf{src1}\%\mathbf{src2}$
MOD_OP	mod	BASE_FORM	$\mathbf{dst} = abs(\mathbf{src1}\%\mathbf{src2})$
AND_OP	and	BASE_FORM	Bit-wise AND
IOR_OP	ior	BASE_FORM	Bit-wise inclusive OR
XOR_OP	xor	BASE_FORM	Bit-wise exclusive OR
ASR_OP	asr	BASE_FORM	Signed shift right
LSL_OP	lsr	BASE_FORM	Unsigned shift right
LSR_OP	lsl	BASE_FORM	Unsigned shift left
ROT_OP	rot	BASE_FORM	Rotate value in **src1** register left (positive value) or right (negative value) by the amount specified in the **src2** register
SEQ_OP	seq	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} == \mathbf{src2})$
SNE_OP	sne	BASE_FORM	$\mathbf{dst} = (\mathbf{src1}! = \mathbf{src2})$
SL_OP	sl	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} < \mathbf{src2})$
SLE_OP	sle	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} <= \mathbf{src2})$
Branch and jump instructions			
JMP_OP	jmp	BJ_FORM	Unconditional jump: **goto target**
BTRUE_OP	btru	BJ_FORM	Branch if true: **if (src1) goto target**
BFALSE_OP	bfls	BJ_FORM	Branch if false: **if (!src1) goto target**
Miscellaneous			
LDC_OP	ldc	LDC_FORM	Load a constant value
CALL_OP	call	CALL_FORM	Call a procedure
MBR_OP	mbr	MBR_FORM	Multi-way branch
LABEL_OP	lab	LABEL_FORM	Label pseudo-instruction

Table 2.1: Valid Simple-SUIF instructions

```
simple_op opcode;          /* the opcode */
simple_type *type;         /* type of the result */
struct simple_instr *next; /* ptr to next instruction */
struct simple_instr *prev; /* ptr to previous instruction */

union u;    /* the variant part of the union is determined
               by the result of simple_op_format(opcode) */

  /* BASE_FORM */
  struct base {
     simple_reg *dst;  /* destination */
     simple_reg *src1; /* source 1 */
     simple_reg *src2; /* source 2 */
  }

  /* BJ_FORM */
  struct bj {
     simple_sym *target; /* branch target label */
     simple_reg *src;    /* source register */
  }

  /* LDC_FORM */
  struct ldc {
     simple_reg *dst;    /* destination */
     simple_immed value; /* immediate constant */
  }

  /* CALL_FORM */
  struct call {
     simple_reg *dst;   /* return value destination */
     simple_reg *proc;  /* address of the callee */
     unsigned nargs;    /* number of arguments. */
     simple_reg **argsl /* array of arguments */
  }

  /* MBR_FORM */
  struct MBR {
     simple_reg *src;      /* branch selector */
     int offset;           /* branch selector offset */
     simple_sym *deflab;   /* label of default target */
     unsigned ntargets;    /* number of possible targets */
     simple_sym **targets; /* array of labels */
  }

  /* LABEL_FORM */
  struct label {
     simple_sym *lab; /* the symbol for this label */
  }
```

Figure 2.3: **simple_instr** structure used to represent a Simple-SUIF instruction

As you can see from Figure 2.3, the **simple_instr** structure contains a member, **u**, which is a union of many other structures[3]. Each of the structures in the union represents a different instruction format in which all the Simple-SUIF instruction opcodes map to. The Simple-SUIF library contains support routines to help the programmer deal with this data structure. Appendix A provides more information on how to use and interact with Simple-SUIF.

[3]A union is used to save memory since an instruction can only be one type of instruction format.

2.5 Summary

In this chapter, three retargetable compiler systems were examined for use as the compiler base for the VSSC framework. The stengths and weaknesses of each were presented. At one end of the spectrum is GCC, a large, slow, monolithic compiler with very good optimization routines. At the other end of the spectrum is `lcc`, a small, fast, non-optimizing compiler whose compact design makes it difficult to expand. The design and organization of these compilers would not fit well in the VSSC framework.

The features of the SUIF compiler framework fulfill the design goals of the VSSC framework far better than those of GCC or `lcc`. The modular design of the SUIF compiler framework allows it to be integrated easily into the VSSC framework. Using the SUIF intermediate format, VSSC has ability to perform optimizations in any order. Finally, incorporating the Simple-SUIF package allows VSSC to be used by both researchers and students.

Chapter 3

Compiler Visualization Tools

Due to recent advances in GUI technology, graphical user interfaces are now becoming easier to create. This allows a program that was previously text-based to have a graphical front end to improve user interaction. A graphical user interface allows compilicated information to be presented more understandable manner. A compiler is a good example of a text-based program that can benefit from a graphical user interface. Such an interface would allow the compiler to display information of interest to those debugging a component of the compiler or learning about the compilation process.

Perhaps an important piece of information that should be displayed in the graphical user interface of a compiler while the back end is executing is the various data structures used in code transformations. These structures can be displayed to illustrate both the analysis and transformation algorithms required to perform a particular optimization. In order to display various types of graphs, a tool to handle the graph drawing capabilities is needed. This chapter first discusses various graph drawing tools, which are an important component of most compiler visualization tools, since graphs are the primary data structures in compilers. The second part of this chapter is a discussion of several compiler visualization projects similar to VSSC.

3.1 Graph Drawing Tools

The graph drawing process is complicated. There is an large amount of research being done in this field [3] and many algorithms have been designed to draw graphs that look aesthetically pleasing. As a result of this research, many graph drawing tools exist to demonstrate the feasibility of these graph drawing algorithms. Such tools include: **ffgraph** [15], **daVinci** [16], $Graph^{Ed}$ [19], and `DOT` [21]. Most tools are stand-alone programs, while others such as `DOT` can be integrated into an existing GUI such as Tcl/Tk [28] [40].

There are several other graph drawing tools that are more specialized in their functionality in that they are used to draw many of the data structures present in compilers: flow graphs, syntax trees, call graphs, and data dependence graphs. Two such tools are `dflo` [44] and **VCG** [32]. **VCG** is used to graphically display typical data structures found in a compiler, while `dlfo` can be used to solve data-flow equations.

3.1.1 VCG

Textual representations of compiler data structures, such as trees or graphs, can often be confusing or unreadable. **VCG** shows trees and graphs in a natural way that allows powerful debugging of the internals of a compiler and the examination of the effects of transformations on the intermediate representation.

The specification of a graph is supplied to **VCG**, which then assigns horizontal and vertical positions to each node and computes splines for the edges in such as way that the edges do not overlap with nodes. Constructed graphs can be *folded* allowing unimportant parts of the graph to be hidden, while important components can be shown in more detail. The output of the constructed graph can viewed using a self-contained X-windows tool or saved as a postscript file.

3.1.2 `dflo`

`dflo` [44] is a tool that inputs: a description of a flow graph, the variables assigned and expressions computed in each flow graph node, and a system of data-flow equations. It then solves the data-flow equations and allows the user to interactively view the results. The flow graphs generated by `dflo` look very nice and the data-flow equations are easy to construct. Figure 3.1 shows the `dflo` data-flow equations to compute liveness.

```
{ LIVE edge - [0] = any succs( LIVE ) * TRANSUP + EXPOSEUP;
  LIVE.in node - [1] = LIVE.out * TRANS + EXPOSEUP ;
  LIVE.out node - [1] = any succs( LIVE.in ) }
```

Figure 3.1: `dflo` data-flow equations to compute liveness

Unfortunately, `dflo` uses the commercial Motif graphical user interface as its GUI front end. It therefore can not be incorporated into VSSC, since a design goal of VSSC is to only use freely available software.

3.1.3 DOT

Neither **VCG** nor `dlfo` would integrate well into the VSSC framework. These tools are designed to be standalone programs, which are difficult to incorporate into a larger package like VSSC. We chose to integrate the `DOT` [21] package with VSSC to implement the graph drawing capabilities VSSC provides.

`DOT` is a general graph drawing tool which draws directed graphs using a four-step algorithm [17]:

1. Assigns discrete ranks to nodes. These ranks determine the Y coordinates in the final drawing.

2. Orders nodes within ranks to avoid crossings.

3. Assigns X coordinates to nodes while keeping edges short.

4. Routes edge splines between nodes that have edges.

The graphs produced by `DOT` are well suited to display the graph data structure information necessary in a compiler. Figure 3.2 shows a sample flow graph for an implementation of the `quicksort` algorithm. The drawing and layout algorithms used in `DOT` are able to generate the graphs fast enough to support interactive GUIs. A detailed explanation of `DOT` and how it is integrated into the VSSC framework is presented in Section 4.3.2.

3.2 Visual Compiler Tools

Previously there were very few compiler visualization tools or visual compilers. Compiler systems tended to be text-based and not designed for use with a graphical

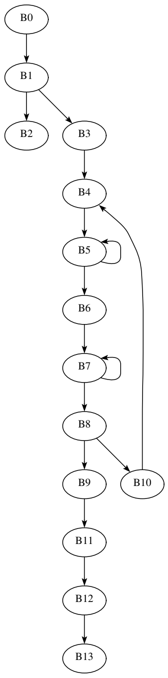

Figure 3.2: `DOT` generated flow graph for `quicksort` algorithm

user interface (GUI). However, more recent compiler systems such as SUIF have been designed with modularity in mind, diminishing the amount of work needed to add a GUI to the system. Several compiler visualization tools that influenced the design of VSSC are presented in this section.

3.2.1 Visual SUIF Browser

One the many tools that have been written for SUIF include a visual SUIF browser [24]. Here is the official description [41]:

> The Visual Browser is a graphical user interface, which runs under X, for browsing through SUIF intermediate representation code. It can open multiple windows for SUIF files, source files, and output C code, and clicking on an object in any of those windows will highlight

the corresponding piece of code in the other windows. Other features include the ability to search for objects with given properties, filter out details which are not currently interesting, and collapse and expand the representation of internals of given nodes.

While this tool acts more as an information browser and is not relevant to the purpose of VSSC, it provides a similar functionality in that it displays the current state of the SUIF intermediate code. Like VSSC, it is also uses Tcl/Tk for its GUI interface.

3.2.2 UW Illustrated Compiler

The University of Washington developed the UW Illustrated Compiler (*icomp*) [2], which lets a user interactively browse through textual and graphical views of control and data structures during the compilation of a program. Almost every part of the compilation process can be viewed including: lexical analysis, parsing, semantic analysis, and code generation. *icomp* uses a construct called *hookpoints*, which are similar to breakpoints, but instead of stopping the execution of the program at a hookpoint, the *icomp* display updates windows that have changed since the last hookpoint was executed. Hookpoints are used to synchronize the illustration of the program with the state of the compiler.

The main purpose of *icomp* is to illustrate the compilation process to undergraduate students. It provides no interface that researchers can use to develop new components easily and no optimizations can be performed. While *icomp* may seem to have limited features, it was one of the first visual compiler tools written and served its purpose well. Students who used the system provided positive feedback regarding its use and its presentation of information.

3.2.3 xvpodb

The tool that most closely matches the goals of VSSC is `xvpodb`, designed by Micky Boyd and David Whalley of Florida State University. *xvpodb* is a graphical optimization viewer for the *vpo* optimizer [4]. It had the following design goals [5]:

- The program should appear in a easily readable display that is automatically updated each time the data structure is changed.
- Indicate the exact portions of the representations that were altered during a transformation.
- Allow data breakpoints to be used.
- Provide the capability to examine the effect of transformations by processing them in forward or in reverse order.

xvpodb runs as a separate process and communicates with *vpo* via sockets. *vpo* sends to a *xvpodb* process messages that describe the changes to make to the RTLs being displayed. Figure 3.3 shows the main window of *xvpodb*. The RTLs are displayed in their basic blocks with arrows illustrating the flow graph. Buttons at the bottom of the screen allow the user to step forward or backward in the current transformation.

The main data structures used by *xvpodb* are the *Optimization List* and the *Screen List*. The *Optimization List* is a list of messages received from the optimizer that describe that changes to make to each RTL as well as information on what to

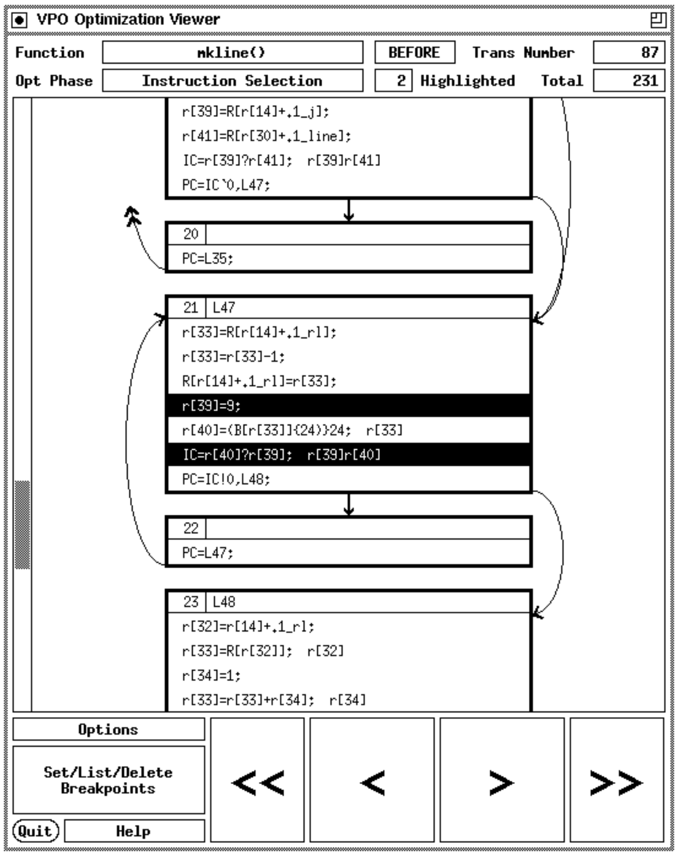

Figure 3.3: Main window of *xvpodb* application

display when stepping in reverse. The *screen list* contains a list of what information is currently being displayed on the screen as well as the current state of each RTL.

xvpodb and VSSC are very similar in their functionality and design goals. However, there are several differences. First of all, *xvpodb* does not allow the user to control the optimizations; it can only be used as a viewing tool, while VSSC was designed to give the user full interactive control of the compilation process. Secondly, *xvpodb* does not allow the user to back out of a transformation, a function that VSSC provides. This ability to undo transformations allows a user to remove the effects of a transformation that turned out to be inappropriate or ineffective. The authors of *xvpodb* claim that adding this functionality would not be difficult for them, but, they have not yet implemented it. One functionality that *xpodb* provides that VSSC does not yet provide is the ability the backstep within a transformation. Both systems allow the user to step forward through a transformation, but *xvpodb* also allows the user to step backwards within a transformation. VSSC allows backstepping at the granularity of a transformation, but not at each step of a transformation.

3.3 Summary

An important component of a compiler with a graphical user interface is the ability to display information about internal data structures used within the compiler. Various types of graphs are used in the back end of a compiler. Rather than implement complicated graph drawing algorithms to draw these graphs, VSSC uses an existing tool that integrates well in the VSSC framework. This chapter discussed several graph drawing tools that specialize in drawing the types of graphs common in compilers. Information was then presented describing previous work in the area of compiler visualization tools. The compiler visualization tools described were: the Visual SUIF Browser, the UW Illustrated Compiler, and *xvpodb*.

The field of compiler visualization tools is relatively young, but with the advent of powerful, easy-to-use GUI languages such as Tcl/Tk, more tools should become available.

Chapter 4

VSSC Design and Implementation

The Visual Simple-SUIF Compiler (VSSC) package is designed to provide an interactive framework that facilitates the development of compiler optimizations. This chapter discusses the various design goals of the VSSC framework and how they were met in the actual implementation. Supporting tools that were used and how they were integrated with the VSSC framework are also discussed. The chapter concludes with a description of the VSSC interface and a sample optimization that demonstrates how an optimization designer would register an optimization using the VSSC package and the SUIF compiler.

4.1 Design Goals

The overall goal in the design and implementation of the VSSC framework is to provide an interface to allow the study of optimizations. The following characteristics are desirable in this interface:

- To be as extendible as possible, the framework should allow new transformations to be added easily.

The VSSC library includes an interface in which any number of transformations are "registered" with the VSSC compiler that is generated.

- The ability to specify the order in which transformations are applied gives the VSSC user the ability to apply transformations in any order and as many times as desired. In order to implement such a feature, a suitable compiler is needed that supports these abilities.

  The VSSC GUI allows the user to select from a menu which transformation to perform next. The underlying compiler used by VSSC is SUIF, which allows transformations to be applied in any order. Section 2.4 describes the SUIF compiler in more detail.

- Support for backing out or *undoing* a transformation. This requires the ability to revert back to a previous version of the intermediate representation. This action undoes any changes made by the current transformation in progress or the previously completed transformation. This characteristic can be extremely helpful in several situations: comparing two different implementations of the same optimization and evaluating different transformation orderings.

  For example, an optimization can be performed and the results observed. Then that transformation is undone and another implementation of the same transformation is performed. The user can then compare the results of the two transformations and determine whether the two implementations of the same transformation produced the same resulting intermediate code.

  The ability to undo transformations can also help researchers studying the phase-ordering problem. The researcher would perform an ordering of transformations and see the result. All those transformations would then be undone and a different ordering of those transformations would be performed. The researcher could then determine if the ordering of the transformations resulted in different intermediate code and which orderings result in more optimized intermediate code.

An undo feature is also useful whenever, during one of the steps of a transformation, the user notices that the transformation incorrectly modifies part of the intermediate representation. At that point, the transformation can be undone before it finishes, reverting the intermediate representation back to its state before the current transformation. VSSC allows the user to undo a transformation at any point. The user can even undo multiple transformations, reverting back as far as the original intermediate code.

- The intermediate format used should be simple, easy to read, and identifiable. VSSC uses the Simple-SUIF interface which provides such an intermediate format. Simple-SUIF is described in more detail in Section 2.4.3.

- VSSC should provide the user a graphical interface capable of displaying information related to the code transformation (in this case, intermediate code and graphs) in an aesthetically pleasing manner. Such an interface is generally easier to use. No typing is necessary; just point and click.

  The interaction between the VSSC compiler and its user is completely graphical. The GUI is implemented using Tcl/Tk [28] [40]. Further information about Tcl/Tk is presented in section 4.3.1 below.

- VSSC should facilitate classroom instruction of compiler optimizations. Very few tools exist today that can be used to *teach* students compiler optimizations graphically. To be an effective teaching tool, the users should be able to follow an optimization at their own pace. VSSC provides a *stepping* functionality that allows the user to *step* through an optimization.

- To be widely used as a tool in the academic research arena, VSSC should be based only on freely available software. Table 4.1 lists the software components integrated with VSSC and where they can be obtained. These components are discussed in Section 4.3 below.

GNU C/C++	`ftp://prep.ai.mit.edu/pub/gnu/gcc-2.7.2.tar.gz`
SUIF	`http://suif.stanford.edu/` `ftp://suif.stanford.edu/pub/suif/basesuif-1.1.2.tar.gz`
Simple-SUIF	`ftp://suif.stanford.edu/pub/suif/simplesuif-1.0.0.beta.1.tar.gz`
Tcl/Tk	`http://www.sunlabs.com/research/tcl/` `ftp://ftp.sunlabs.com/pub/tcl/`
DOT/tcldot	`http://www.research.att.com/sw/tools/reuse/`

Table 4.1: VSSC components and where to find them

4.2 Organization of VSSC Framework

The VSSC framework is implemented as a library that is linked in with the optimization writer's code. This library, along with the libraries for SUIF, Simple-SUIF and `tcldot`, contains everything that is needed to construct a VSSC compiler. The VSSC library acts as a manager of the various sub-libraries it uses. It takes the strengths of each sub-library and works around their implementation weaknesses to produce a tightly integrated system. Figure 4.1 depicts those components of the VSSC framework and how these components interact. Information on how to obtain and install these packages can found in Section A.4.3 of Appendix A.

The next section describes the components of the VSSC framework and how

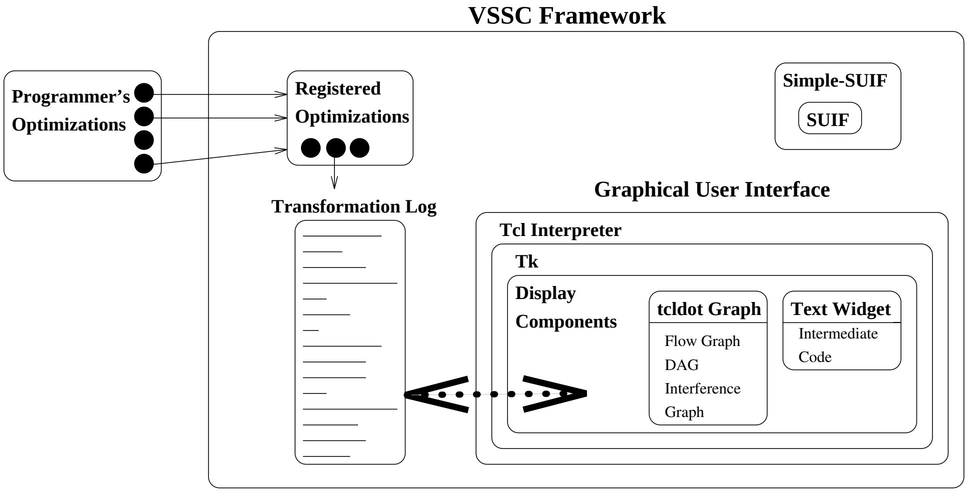

Figure 4.1: Internal organization of VSSC framework

they integrate with each other.

4.3 Implementation of VSSC

Each of the components in Figure 4.1 provides a solution of one or more of the design goals proposed in Section 4.1. The core of the VSSC framework is implemented in C/C++. It would be possible to implement the functionality provided by VSSC in any high-level programming language that can be linked with the existing C/C++ libraries[1].

4.3.1 Graphical User Interface

A primary component of the VSSC framework is the graphical user interface. This component implements several of the design goals outlined previously. The GUI allows the VSSC compiler to be "interactive" as well as providing the ability to display the intermediate code and various graphs simultaneously. Visual changes

[1] The SUIF, Simple-SUIF, Tcl/Tk, and `tcldot` libraries are all C or C++ libraries

to the intermediate code and graphs can be animated. The GUI is also the key component that allows the user to select which transformations to perform and in what order to execute them.

Tcl/Tk [28] [40], developed by John Ousterhout at the University of California, Berkeley, is a scripting language. The Tcl/Tk graphical user interface is used to implement VSSC's GUI. It is a simple and portable toolkit that can be easily integrated with C/C++ code. Tcl is the scripting language itself while Tk is an extension to Tcl that provides X windows GUI development capabilities. Together, they have become very popular for three reasons:

- Because the language is interpreted, there are no waits for long compilations. Code can be tested immediately, yielding fast development cycles.

- Tk provides a high-level interface to the complicated X windows system. Simple user interfaces can be created with just a few lines of code.

- Simplification of the development of the user interface allows the programmer to concentrate more on the internal core of the application.

The VSSC framework includes an embedded Tcl/Tk interpreter, which provides the rest of VSSC library direct communication with the display and access to the display components. Figure 4.2 shows the VSSC window displaying the intermediate representation and flow graph of `quicksort`.

The VSSC GUI contains three main areas over which the user has control. Figure 4.3 shows the first area (Current Status) which resides in the top left corner of the VSSC interface. This area simply acts as a general information area. Since VSSC allows the user to step through the optimization (much like a *step* in a

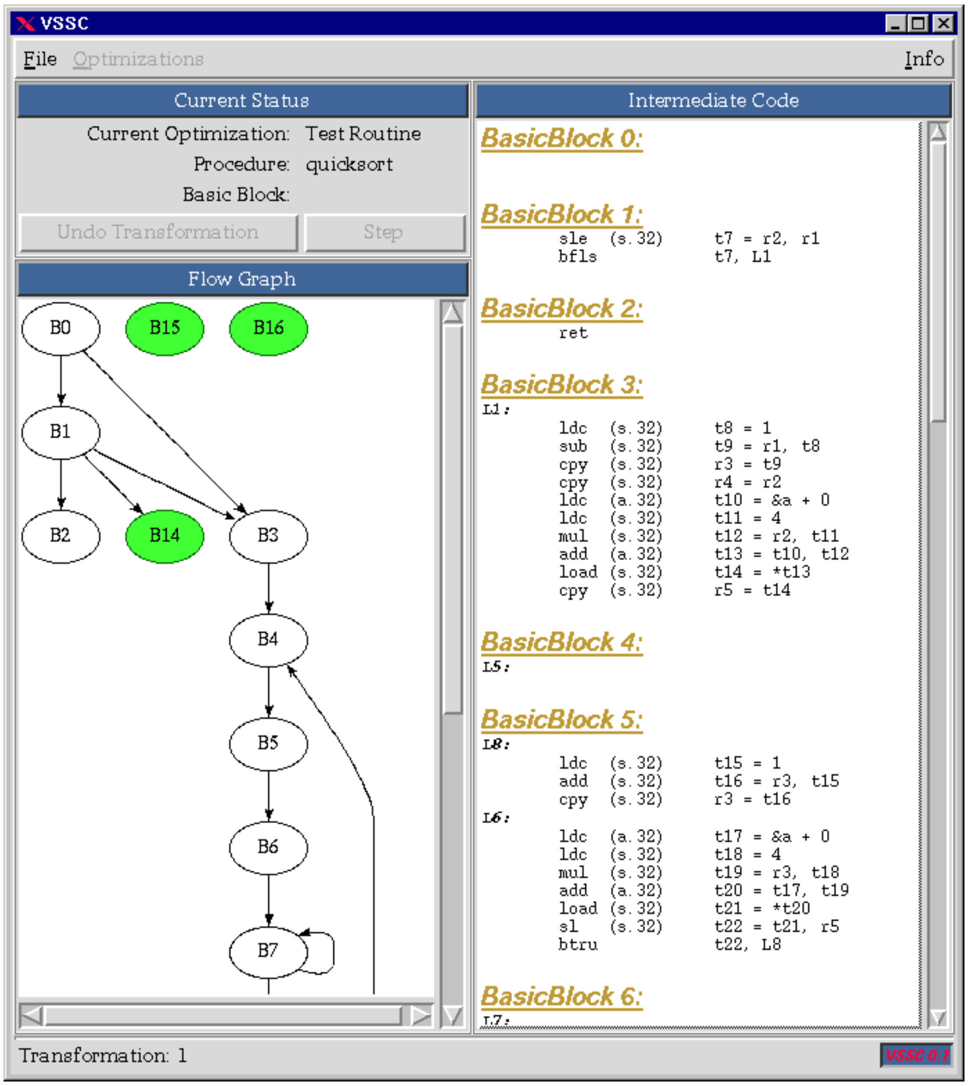

Figure 4.2: Screenshot of VSSC compiler

debugger), this first area contains two buttons that allow the user to be able to *step* forward or *undo* an entire transformation. During each step, any number of actions (ie adding/deleting instructions/graph nodes) can occur. The optimization writer decides what happens during each step.

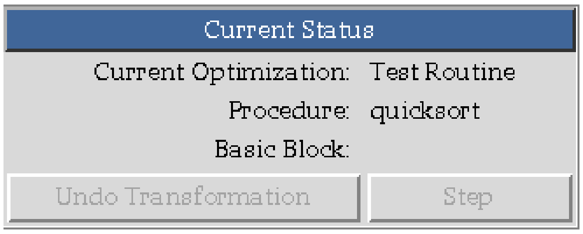

Figure 4.3: Current status component

The area below the Current Status area contains a graph widget. In this area the optimization writer can create graphs. The most common types of graphs that can be created are flow graphs, directed acyclic graphs (DAGs), and interference graphs[2]. Each node in the graph widget can have arbitrary data associated with it. This data is displayed when the user clicks on the graph node with the left mouse button. Clicking in the box that contains the data hides it. Figure 4.4 shows a sample graph component.

The last area, located on the right hand side of the screen, is a text widget containing the Simple-SUIF intermediate code. The code presented in this widget is usually contained within basic blocks (as in Figure 4.5).

The VSSC compiler contains several menus. The *File* menu provides the ability to quit the compiler, dump a copy of the graph currently being displayed to a file, and enter Tcl/Tk commands directly. This last feature is a debugging tool for extending the VSSC GUI component. The *Optimizations* menu lists the trans-

[2]Graphs of these types will be demonstrated in Chapter 5.2

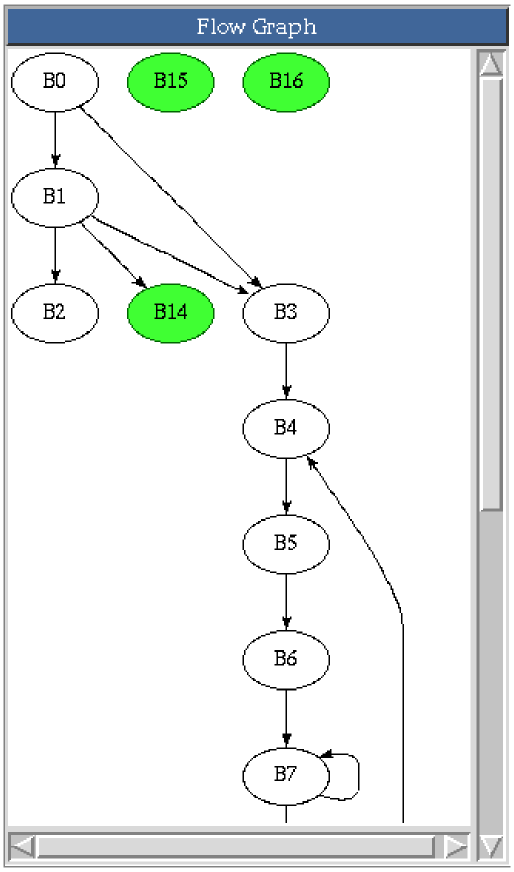

Figure 4.4: Graph component

formation routines registered and available in this VSSC compiler. Selecting an entry in this menu causes that transformation to occur on the current state of the intermediate code. During the execution of the transformation, this menu is unavailable, because a new transformation cannot be started while another one is in progress. Once the current transformation has completed, however, this menu is again available. If a transformation is *undone*, this menu also becomes available, since this action terminates the current transformation. Section 4.3.6 describes the *undo* feature in more detail.

The *Display Components* component of the VSSC framework manages every-

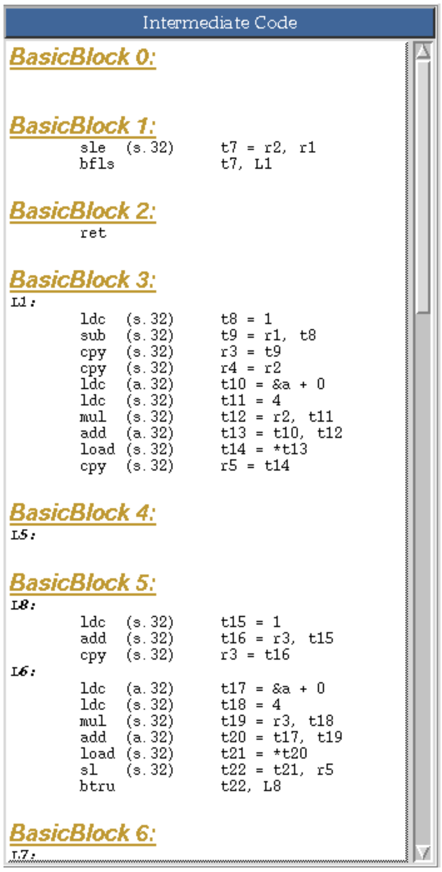

Figure 4.5: Intermediate Code component

thing that is displayed on the screen and updates the display depending on commands it receives from each entry in the transformation log. The transformation log is described in detail in Section 4.3.5.

4.3.2 Incorporating `tcldot` into VSSC

A graphical user interface for a compiler tool should be able to display the types of graph data structures that are commonly used in compiler tools. Applications of graph visualization are discussed in [27], while the issue of graph visualization from the viewpoint of compilers is discussed in detail in [33].

Rather than implement custom graph drawing algorithms within the VSSC GUI, we chose to integrate an existing graph drawing package in the VSSC framework. We chose to use `tcldot` [22], which is part of a larger package, `DOT` [21], developed by Eleftherios Koutsofios and Stephen C. North at AT&T Bell Laboratories.

`tcldot` is a version of `DOT` that can be used to produce graphs in a Tk canvas widget. VSSC embeds `tcldot` within its framework and uses it to draw graphs on a canvas located in the Graph Component of the GUI. The `DOT/tcldot` API [22], part of which is shown in Table 4.2, includes many commands such as creating and destroying graph nodes and edges between nodes. The VSSC library provides a `Graph` class that acts as a *wrapper* for `tcldot` and allows the programmer to control the information being displayed in the graph using C/C++ code. VSSC's graph class also allows the programmer to annotate information with each node. This information is displayed in a pop-up box whenever the user clicks on the node in the graph.

4.3.3 SUIF

The SUIF component provides two functions. First, it converts the C input file into the SUIF intermediate format. This action is performed by calling the SUIF compiler, which performs all the front end translation. This functionality is important in meeting the design goal of using a simple intermediate format. After this first translation process, the Simple-SUIF component of VSSC converts the the intermediate format generated by the SUIF compiler into Simple-SUIF. This process will be described in Section 4.3.4.

The other function provided by the SUIF library is the use of data structures

Commands	Purpose
dotnew *graphtype*	Creates a new graph of type `graph`, `digraph`, `graph strict`, or `digraph strict`. A graph handle is return that is used on all future references to this graph.
dotread *fileHandle*	Reads a graph from a file handle.
graphHandle **dotwrite** *fileHandle*	Saves a graph to a file in one of several formats.
graphHandle **addnode**	Adds a node to a graph. Returns a handle for that node.
graphHandle **addedge**	Adds an edge to a graph. Returns a handle for that edge.
handle **delete**	Deletes a node, edge, or entire graph.
handle **setattributes**	Set the attributes for a node, edge, or graph.
graphHandle **layout**	Computes layout of nodes and edges.
graphHandle **render**	Returns list of Tk canvas commands which can be `eval`ed to draw the graph in a canvas.

Table 4.2: Some commands in the `tcldot` API

to help implement the other components of the VSSC framework.

Writing compiler optimizations (or any large project) requires that you manage complex data structures. Commonly used data structures in compiler optimizations and compiler tools include lists, trees, arrays, bit sets, and graphs.

There are many popular data structure libraries that work well. One such library is LEDA [26]. LEDA was considered for use in VSSC during the design phase, but it was rejected because it would add another package to the list of packages VSSC already requires.

Fortunately, the SUIF libraries include most of the commonly used data structures needed in compiler tools. The VSSC framework uses many of the generic data structures provided by SUIF. These data structures are described in more detail in section A.5.5 of Appendix A and Chapter 11 of the SUIF Library Documentation

[37].

4.3.4 Simple-SUIF

One of the major design goals of the VSSC framework is to use an intermediate format that is simple, easy to read, and familiar looking. As described in section 2.4.3, Simple-SUIF acts as a *wrapper* for SUIF by providing a simplified interface to the intermediate format generated by the SUIF compiler. When using Simple-SUIF, the SUIF intermediate format remains the same internally, but differs in the way the programmer interacts with it.

The Simple-SUIF component in the VSSC framework provides several different important functions. First it converts the SUIF format generated by the SUIF component into the Simple-SUIF format. This is done by making several calls to the SUIF program, `porky`, which performs several types of code transformations. `porky` is used to remove all of the high-level construct information from the SUIF intermediate format (eg. loops). The result of this transformation is a more low-level intermediate format that can be converted into Simple-SUIF instructions by the Simple-SUIF component.

The second job of the Simple-SUIF component is to manage how the Simple-SUIF instructions are used by the programmer writing the optimization. When a registered optimization is selected to run, this component uses the Simple-SUIF library to read in the intermediate format from a file, convert it to Simple-SUIF using `porky`, and then passes a list of instructions to the registered optimization routine. This routine can then do whatever it wants with the list. The only requirement is that the routine return a doubly-linked list of Simple-SUIF instructions back to the Simple-SUIF component. The returned linked-list is then written back

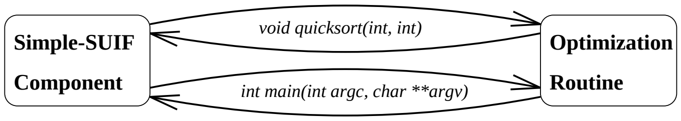

Figure 4.6: Simple-SUIF component passing lists of Simple-SUIF instructions one at a time for each function to an optimization routine.

out to a file in SUIF format. This file is re-read when a subsequent optimization is performed. If a structurally correct doubly-linked list is not returned, the entire framework will exit with an error. It is the programmer's responsibility to ensure that a correct list is returned.

If the intermediate code contains more than one function, each function consisting of a list of Simple-SUIF instructions is passed to the selected optimization routine sequentially as shown in Figure 4.6. Simple-SUIF processes one function to completion before the next one is processed. This design simplifies the structure for the compiler writer but severely limits the ability to perform interprocedural analysis[3].

Finally, the Simple-SUIF library includes routines to help with the management of Simple-SUIF instructions. It includes routines for creating, removing, and determining the format of Simple-SUIF instructions.

4.3.5 Transformation Log

Each transformation is decomposed into a set of *steps*. These steps are defined by the optimization writer and any number of actions can occur between steps. Typically though, only one or two major actions (ie adding or deleting an instruction or graph node) are performed during each step. All the steps for a single transfor-

[3]Though standard SUIF provides this ability through the use of *annotations*, Simple-SUIF does not.

mation are stored in one *transformation log* which acts as a *recording* of what to display in the graphical user interface. This log, implemented as a linked-list, is shown in Figure 4.7. It is similar in design to the *Optimization List* used in `xvpodb` [5] except that each log entry can have more than one event.

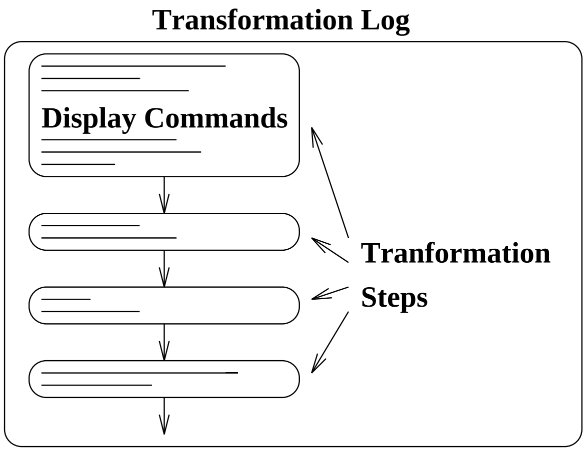

Figure 4.7: Internal organization of the transformation log

When the user starts a transformation within a VSSC compiler, the transformation does not occur in real-time but executes to completion in the background while all VSSC transformation actions and defined steps are stored in a new transformation log. The VSSC GUI *plays* this log by executing its display actions for the user. When the user first selects the *step* button, the first step in the transformation log is performed; usually, the first step involves displaying the basic blocks of intermediate code in the text widget. Future selections of the *step* button cause subsequent steps in the transformation log to be processed. When the last step in the transformation log is executed, the transformation is finished and a pop-up window appears notifying the user. At this point, the user has to ability to perform another transformation on the current state of the intermediate code or undo the effects of this last transformation and revert back to the previous version of the intermediate code.

4.3.6 Undo Transformation

One of the design goals of VSSC is to provide the ability to *undo* transformations. This capability yields benefits in two situations. First, it can be helpful when analyzing two different implementations of the same optimization (e.g.two different implementations of dead code elimination). One optimization can be performed and the results observed and then that transformation can be undone and another implementation of the transformation can be performed. The user can then compare the results of the two transformations. Another situation in which undoing transformation is extremely valuable is when a researcher is analyzing the phase-ordering problem. The researcher can perform an ordering of transformations and see the result. All those transformations can be then undone and a different ordering of those transformations can be performed producing a possibly different result than the first ordering.

This feature is easy to incorporate into the VSSC framework because of the flexibility on the overall design. Since the SUIF format, and therefore Simple-SUIF, is always stored in a file, VSSC keeps the results before and after each transformation in a SUIF file. This organization acts as a *stack* of transformations. When the user requests to undo a transformation (which can happen at any time during the use of a VSSC compiler, even during the middle of a transformation), the current intermediate code is forgotten and the previous version of the intermediate code is *popped* off the stack. When the user selects the next transformation to execute, the Simple-SUIF component of the VSSC framework uses the previous version of the intermediate format. A VSSC compiler allows the user to undo transformations all the way back to the beginning, since the VSSC framework remembers all versions of the intermediate code during a single execution of a VSSC compiler.

4.4 Programmer Interface

The VSSC framework is a library that is linked in with the optimization writer's code. When this library is linked in, the resulting executable is a new VSSC *compiler*. This compiler gives its users access to those code transformations/optimizations that were written by the optimization writer and subsequently *registered* with the VSSC library. The VSSC compiler only knows about those code transformations that have been *registered* with the VSSC library. The process of registering requires the optimization writer to include code that calls a C function that registers a function to be called whenever the VSSC compiler user wishes to perform that transformation. This process is similar to a *callback* function, which is commonly used in programming with graphical user interfaces. An example of this registration process can be found in section 4.5.

The typical organization of a VSSC compiler source code consists of a `main` function which registers the various optimizations and a set of optimization routines. The `main` function concludes with a call to `vssc_init_suif`. This last procedure call never returns, so the programmer should not include any code after it, and performs a number of tasks.

First, it initializes the SUIF and Simple-SUIF subsystems. This includes converting the C source code being compiled into a Simple-SUIF format. Secondly, `vssc_init_suif` parses the command-line flags shown in Table 4.3[4]. A VSSC compiler has the ability to run in two different modes. The first mode is the normal mode that pops up a graphical user interface and allows the user to perform the transformations available in the VSSC compiler in any order. The other mode allows the VSSC compiler to run without a graphical user interface. When it is run without a graphical user interface, the user must specify with command-line flags which transformations to perform and what order to perform them in. This text-only mode does not allow the user to undo transformations. Also, since this mode is non-graphical, very little information is displayed as the transformations are performed. The results of each transformation are stored in a separate file. The user can then use the Simple-SUIF command, `printsimple`, to look at each transformation result and track the changes made in the source code from it original form to its resulting form[5]. Therefore, when parsing the command-line flags, the VSSC system needs to determine the mode in which to execute, and if run in non-graphical mode, it needs to determine which transformations the user wants to perform and whether or not these transformations exist in the VSSC compiler. If being run in graphical mode, `vssc_init_suif` starts up the graphical user interface and initializes its components.

To access the VSSC API, the programmer needs to include a single header file for each package: VSSC, SUIF, and Simple-SUIF. The VSSC compiler also needs to be linked with the libraries for SUIF, Simple-SUIF, Tcl/Tk, `tcldot`, and VSSC libraries. Section A.4.6 in Appendix A describes in more detail the sample optimization in the next section as well as instructions on how to download a sample

[4]Example uses of these command-line flags are shown in Section A.5.3 in Appendix A.

[5]These files are also generated when the VSSC compiler is run in graphical mode.

`-v`	Executes the VSSC compiler in graphical mode.
`-O`	When the VSSC compiler is executed in non-graphical mode each `-O` flag specifies what optimization to perform. When this flag is used, the `-v` cannot be used.
`-d`	Executes the VSSC compiler in *debug* mode. This allows a debugger (e.g. GDB) to be used to debug code in a transformation. The compiler can be in either graphical or non-graphical mode.

Table 4.3: Command-line flags accepted by a VSSC Compiler

optimization that includes a SUIF-like Makefile to produce a VSSC compiler with all the required libraries linked in.

4.5 Sample Optimization

In this section, we present sample code that shows a sample optimization and how it gets registered. Calls to `GUI->step()` signify the end of a *step*. VSSC API commands executed between calls to `GUI->step()` are those that are executed during a single *step* within a transformation executed in a VSSC compiler. Figure 4.8 below shows this example code.

4.6 Summary

In this chapter, the design goals of the VSSC framework were presented along with an explanation of how they were implemented. The organization of the VSSC framework was then described followed by an explanation of the framework's components. The SUIF compiler and Simple-SUIF were used to implemented the intermediate format and the ability to selectively apply transformations to the in-

```
#include <stdio.h>
#include <vssc_simple.h>

#include "BasicBlock.h"
#include "FlowGraph.h"

simple_instr *dead_code(simple_instr *inlist, char *procedure_name)
{
   FlowGraph *FG;

   /* Set what the procedure name is and what basicblock we're looking at. */

   GUI->set_procedure_name(procedure_name);
   GUI->set_basicblock_number(BLANK);
   GUI->set_graph_type("Flow Graph");
   fprintf(stderr, "Doing deadcode elimination\n");

   FG = new FlowGraph(inlist, procedure_name);

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   /* Etc. */

   return FG->instructions_head;
}

int main(int argc, char *argv[])
{
   vssc_register_opt("Dead Code Elimination", "deadcode", dead_code);

   vssc_init_suif(argc, argv);
}
```

Figure 4.8: The beginnings of a sample optimization

termediate code. Tcl/Tk was used as the language to implement the graphical user interface. Finally, the `DOT` drawing package was used to allow the GUI to display various types of graph data structures. The chapter concluded with a desciption of the VSSC programmer interface and a sample optimization.

Chapter 5

VSSC Framework Examples

5.1 Introduction

A variety of compiler analysis and transformations as well as a diverse collection of graphs typically found in compiler transformations have been implemented to demonstrate the capabilities of the VSSC framework. This chapter will describe some transformations to show the capabilities provided by the VSSC framework. Section 5.2 shows the various graphs that have been implemented. These include a flow graph, directed acyclic graph, and a register-interference graph. An example in which live-variable analysis is performed followed by dead-code elimination is shown in Section 5.3. Finally, Section 5.4 shows an example in which available-expression analysis is performed followed by copy propagation. The algorithms discussed in this section are described in [1] and [30].

5.2 Example Graphs

Graph data structures are used throughout a compiler. The front end uses a parse tree (an acyclic graph with a single node recognized as the "root") when parsing the source code during syntax analysis. During syntax-directed translation, syntax trees are used to represent language constructs. The syntax tree is traversed in the front end to construct the intermediate representation. The back end of a compiler also uses a variety of graph data structures. This section provides examples that show how three different graph data structures can be used in the VSSC framework to convey information related to the intermediate code.

5.2.1 Flow Graph

A flow graph illustrates the flow-of-control information for an individual function. Each node in the graph represents a basic block[1]. There is a directed edge in the graph from block B_i to block B_j if B_j can immediately follow B_i in some execution sequence. In this scenario, B_j is a successor of B_i while B_i is a predecessor of B_j. Therefore, the directed edges in a flow graph represent edges to successors. It should also be noted that a basic block can have multiple predecessors and multiple successors.

Before a flow graph can be constructed, the basic blocks of a procedure need to be determined. Algorithm 9.1 in [1] described a two-step method:

1. First determine the set of *leaders*, the first statements of basic blocks. The rules used are the following:

[1] A basic block is a maximal sequence of consecutive statements in which flow of control enters only at the beginning of the block and exits only at the end of the block.

(a) The first statement (in the procedure) is a leader.

(b) Any statement that is the target of a conditional or unconditional goto (essentially a control-successor) is a leader.

(c) Any statement that immediately follows a goto or conditional goto statement is a leader.

2. For each leader, its basic block consists of the leader and all consecutive statements after it, up to, but not including, the next leader (or the end of the procedure).

Once the basic blocks have been found, the successors of an individual block can be determined by looking at the last statement in the block and determining the possible blocks to which control can flow. In constructing a flow graph, the following simple algorithm can be used:

1. For each basic block found in the previous algorithm, create a node in the flow graph to represent that basic block.

2. For each basic block B_i, determine the successor blocks by looking at the last instruction

   (a) For each successor block B_j, create an edge in the flow graph from B_i to B_j. In the data structure representing B_i, remember that B_j is a successor and in the data structure representing B_j, remember that B_i is a predecessor. This information is used in various data-flow analysis algorithms, including those discussed in Sections 5.3 and 5.4.

```
int main()
{
   int A[] = {4,8,2,7,1,13,19,11,3,0,
              18,12,17,9,16,10,15,14,6,5};
   int i, j, temp;

   for (i=18; i>=0; i--)
   {
      for (j=0; j<=i; j++)
      {
         if (A[j] > A[j+1])
         {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
         }
      }
   }
}
```

Figure 5.1: Example C program of bubblesort

```
Procedure main:
BASICBLOCK 0:

BASICBLOCK 1:
        ldc  (a.32)     t4 = &__tmp_string_0 + 0
        ldc  (a.32)     t5 = &A + 0
        mcpy            *t5 = *t4
        ldc  (s.32)     t6 = 18
        cpy  (s.32)     r1 = t6

BASICBLOCK 2:
L9:
        ldc  (s.32)     t7 = 0
        sle  (s.32)     t8 = t7, r1
        bfls            t8, L6

BASICBLOCK 3:
        ldc  (s.32)     t9 = 0
        cpy  (s.32)     r2 = t9

BASICBLOCK 4:
L7:
        ldc  (a.32)     t10 = &A + 0
        ldc  (s.32)     t11 = 4
        mul  (s.32)     t12 = r2, t11
        ldc  (s.32)     t13 = 4
        add  (s.32)     t14 = t12, t13
        add  (a.32)     t15 = t10, t14
        load (s.32)     t16 = *t15
        ldc  (a.32)     t17 = &A + 0
        ldc  (s.32)     t18 = 4
        mul  (s.32)     t19 = r2, t18
        add  (a.32)     t20 = t17, t19
        load (s.32)     t21 = *t20
        sl   (s.32)     t22 = t16, t21
        bfls            t22, L5

BASICBLOCK 5:
        ldc  (a.32)     t23 = &A + 0
        ldc  (s.32)     t24 = 4
        mul  (s.32)     t25 = r2, t24
        add  (a.32)     t26 = t23, t25
        load (s.32)     t27 = *t26
        cpy  (s.32)     r3 = t27
        ldc  (a.32)     t28 = &A + 0
        ldc  (s.32)     t29 = 4
        mul  (s.32)     t30 = r2, t29
        add  (a.32)     t31 = t28, t30
        ldc  (a.32)     t32 = &A + 0
        ldc  (s.32)     t33 = 4
        mul  (s.32)     t34 = r2, t33
        ldc  (s.32)     t35 = 4
        add  (s.32)     t36 = t34, t35
        add  (a.32)     t37 = t32, t36
        mcpy            *t31 = *t37
        ldc  (a.32)     t38 = &A + 0
        ldc  (s.32)     t39 = 4
        mul  (s.32)     t40 = r2, t39
        ldc  (s.32)     t41 = 4
        add  (s.32)     t42 = t40, t41
        add  (a.32)     t43 = t38, t42
        str             *t43 = r3

BASICBLOCK 6:
L5:
L3:
        ldc  (s.32)     t44 = 1
        add  (s.32)     t45 = r2, t44
        cpy  (s.32)     r2 = t45
        sl   (s.32)     t46 = r1, r2
        bfls            t46, L7

BASICBLOCK 7:
L4:
        jmp             __done8

BASICBLOCK 8:
L6:
        ldc  (s.32)     t47 = 0
        cpy  (s.32)     r2 = t47

BASICBLOCK 9:
__done8:
L1:
        ldc  (s.32)     t48 = -1
        add  (s.32)     t49 = r1, t48
        cpy  (s.32)     r1 = t49
        ldc  (s.32)     t50 = 0
        sl   (s.32)     t51 = r1, t50
        bfls            t51, L9

BASICBLOCK 10:
L2:
        ldc  (s.32)     t52 = 0
        ret             t52
```

Figure 5.2: Simple-SUIF version of bubblesort partitioned into basic blocks

Figure 5.1 shows an implementation of the bubblesort algorithm. The Simple-SUIF version of this code partitioned into basic blocks by the above algorithm is shown in Figure 5.2. It is not uncommon for a small input source file to be represented by many times more lines of intermediate code. Finally, the flow graph representing the flow-of-control between these basic blocks in shown in Figure 5.3.

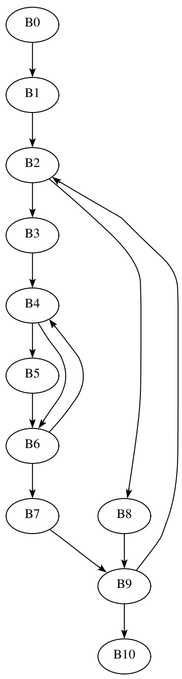

Figure 5.3: Flow graph of basic blocks for bubblesort example shown in Figure 5.2

5.2.2 Direct Acyclic Graph

Directed acyclic graphs (DAGs) are useful data structures in compiler transformations. A DAG usually represents a single basic block and contains information depicting how the value computed by each statement in the basic block is used in subsequent statements of the same block. This information can be used to find common subexpressions (those expressions that are computed more than once) as well as determining which statements in the block could have their value used outside the block. Nodes within a DAG are labeled using the following rules [1]:

1. Each leaf in the graph is labeled by a unique identifier, which is either a constant value such as "4" or "56", a register, or a symbol name.

2. Interior nodes are labeled by an operator symbol. These nodes represent computations with one or two operands. For such a node, there is a directed edge from the node to each node which represents the current value of each operand.

3. Nodes are optionally given a sequence of identifiers for labels. Non-leaf nodes represent values that have been computed. This sequence of identifiers represents those identifiers which have the same computed value. For example, if there is an interior node labeled "$*$" and the current basic block contains several statements each of which computes $2 * 3$, each destination register in these computations will be in the identifier list for the same "$*$" node because each of those registers represents the same computed value.

For an example showing the construction of a DAG, consider the following sequence of Simple-SUIF instructions (basic block #6 in the bubblesort example in Figure 5.2):

```
ldc t44 = 1

add t45 = r2, t44

cpy  r2 = t45

sl  t46 = r1, r2
```

The steps taken in constructing the DAG representing this sequence of instructions (see Figure 5.4) are:

1. `ldc t44 = 1`
   (a) Create a leaf node labeled "1".
   (b) Add "t44" to the identifier list of the newly created node "1".

2. `add t45 = r2, t44`
   (a) Create a leaf node labeled "r2".
   (b) Create a parent node "+" with edges to the children nodes "r2" and "1" ("1" is the node which contains the current value of "t44").
   (c) Add "t45" to the identifier list of the newly created node "+".

3. `cpy  r2 = t45`
   (a) Add "r2" to the identifier list of the node which represents the current value of "t45" (node "+").

4. `sl t46 = r1, r2`
   (a) Create a leaf node labeled "r1".
   (b) Create a parent node "<" with edges to the children nodes "r1" and "+" ("+" is the node which contains the current value of "r2").
   (c) Add "t46" to the identifier list of the newly created node "<".

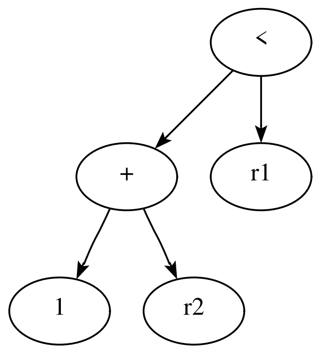

Figure 5.4: Result of DAG construction for basic block #6 in bubblesort example

Slightly more complex DAGs of basic blocks 4 and 5 from the bubblesort example (Figure 5.2) are shown in Figures 5.5 and 5.6. These figures also demonstrate

VSSC's ability to associate arbitrary text with each individual node in a graph. In the graphs shown, the identifier list indicating which registers contain the value computed at this node is displayed when a node is selected[2].

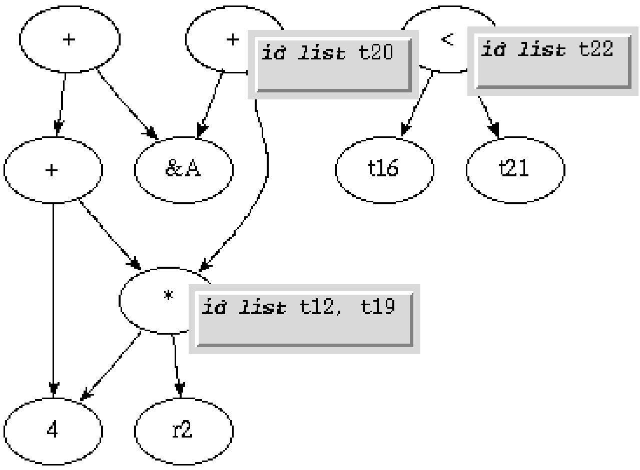

Figure 5.5: Directed acyclic graph for basic block #4 in bubblesort example

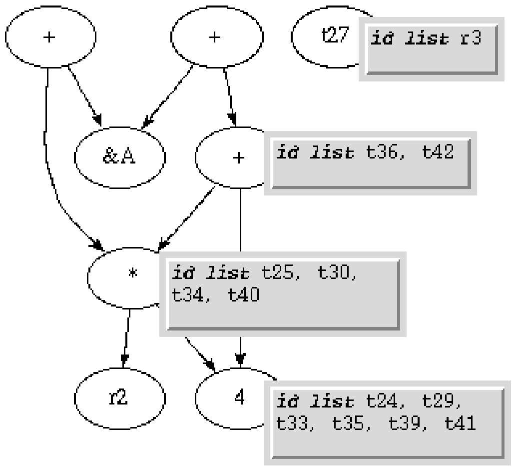

Figure 5.6: Directed acyclic graph for basic block #5 in bubblesort example

In Figure 5.6, the "$*$" node has the registers `t25`, `t30`, `t34` and `t40` in its identifier list. This information indicates that the same expression, $r2 * 4$, is computed

[2]These figures were generated using the VSSC compiler's ability to dump the current contents of the graph to a postscript file.

four times within this basic block. Only the first of these computations is really needed, while the other three are wasteful in computing a value that has already been computed. Common subexpression elimination could now be performed to replace these last three "common subexpressions" with a reference to the register that contains the result of the first occurrence of the expression. This transformation is useful in reducing execution time when the common subexpressions are multiplies or divides, which are normally high latency operations.

5.2.3 Register-Interference Graph

A register-interference graph is used to implement register allocation via a graph-coloring method. The nodes in this graph are symbolic registers and an edge connects two nodes (registers) if one register is live at a point where the other is defined. In order to make these edges, live variable analysis needs to be performed first.

Figure 5.7 shows the register-interference graph for the bubblesort Simple-SUIF code shown in Figure 5.2.

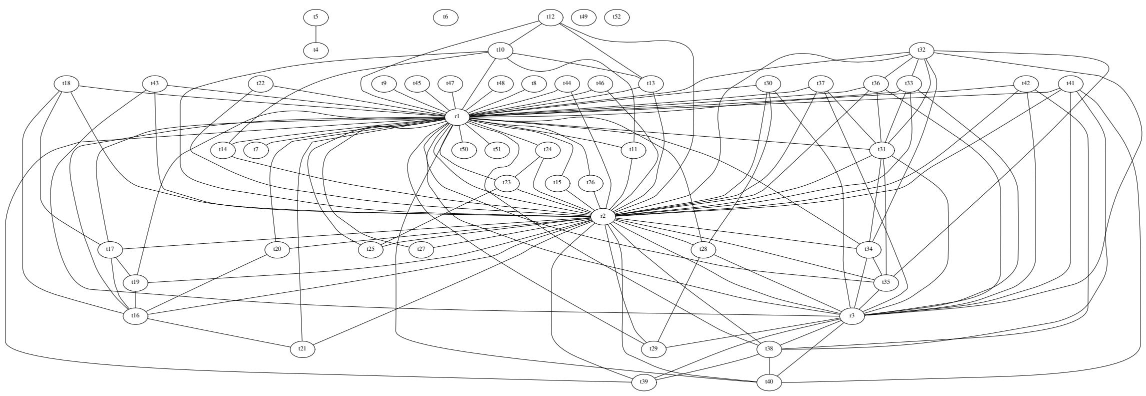

Figure 5.7: Register-interference graph for bubblesort example

5.3 Example Optimization: Elimination of Dead Code

5.3.1 Dead Code Elimination

Dead-code elimination is one of the most common optimizations. Simply stated, dead-code optimization removes code that is *dead*. Code that cannot be reached along any path of execution is considered dead. Also, a statement in the intermediate code is dead if it calculates a result that will never be used. Since the result is never used, one can remove this entire statement without changing the *meaning* of the program. It can determine whether the destination of a statement is dead by performing live variable analysis. A variable that is not live is considered dead. The next section describes how to compute liveness information.

5.3.2 Live Variable Analysis

Performing live variable analysis provides important information about the variables (registers) in the intermediate code. For any point in the intermediate code, live variable analysis can determine for any variable, whether or not that variable is used in any of the possible paths in the flow graph from that point. If there is a future *use* of a variable, then that variable is considered *live* at that point. Otherwise, it is considered *dead* there, because its value will not be used again.

The live-variable analysis algorithm uses bitset data structures to hold liveness information. Each basic block contains the bitsets shown in Table 5.1. The bits in a bitset map directly to the variable with the same number. For example, bit 4 in a bitset refers to register 4.

It is important to note that the Simple-SUIF intermediate representation [36] guarantees that a temporary register is defined only once and it is used only once within the same basic block in which it was defined. For this reason, live-variable analysis only keeps track of Simple-SUIF pseudo registers.

Bitset	**Purpose**
in	The set of variables live at the entrance of the basic block.
out	The set of variables live at the exit of the basic block.
def	The set of variables assigned values in the basic block prior to any use of that variable in the basic block.
use	The set of variables whose values may be used in the basic block before any definition of that variable in the basic block.

Table 5.1: Bitsets used during live variable analysis

Given the *def* and *use* bitsets computed for each basic block, the algorithm[1] in Figure 5.8 will perform live variable analysis computing the *in* and *out* bitsets for each basic block.

```
foreach basic block B
   B.in = Empty Set

while there are any changes to any B.in's
{
   foreach basic block B
   {
      B.out = union of all S.in's for each successor S of B
      B.in = union of B.use and (B.out - B.def)
   }
}
```

Figure 5.8: Algorithm for live variable analysis

Essentially, for each basic block B, variables that are live at the end of B are those that are live at the entrances to some successor block of B. Those variables that are live at the entrance to B are those which have a *use* in B plus those that are live at the exit of B and are not defined (killed) within B. This algorithm propagates liveness information backwards through the flow graph until no new information gets propagated, i.e., stability has been achieved.

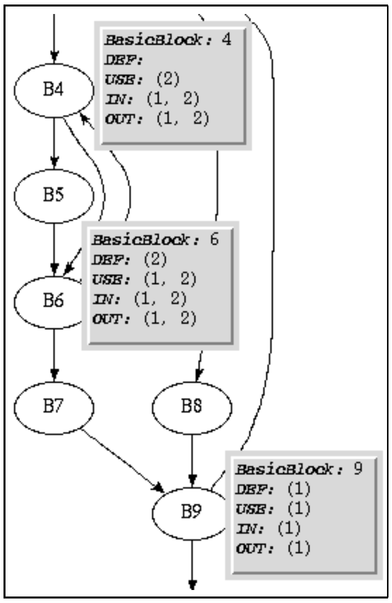

Figure 5.9: Various results after performing live variable analysis

Figure 5.9 shows the final bitset information for several basic blocks in the intermediate representation of the bubblesort example.

5.3.3 Eliminating Dead Code in Bubblesort Example

Figure 5.10 shows a simple version of the dead-code elimination algorithm similar to the algorithm in [30].

More often than not, the source program is not responsible for dead code that appears in the intermediate code. The dead code may actually appear as a result of previous code transformations or by the way the front end converts the source

```
foreach Basic block B
{
   Bitset currentlyLive = B.out
   foreach instruction I in B from last to first
   {
      if I.dest is currentlyLive
      {
         Remove I.dest from currentlyLive
         Add I.src1 to currentlyLive
         Add I.src2 to currentlyLive
      }
      else
         remove instruction I since its destination is dead
   }
}
```

Figure 5.10: Algorithm for the removable of dead code

language into the intermediate representation. An example of this occurred when the bubblesort C code was translated into Simple-SUIF intermediate code by the front end of the SUIF compiler. Basic block #8 of the intermediate representation contains instructions that simply load the constant 0 into register $r2$. Performing dead-code elimination removes these two instructions from the basic block. It is not important to understand why this code was generated in this example is not important. What is important is to understand *why* dead-code elimination decided to remove these two instructions.

Figure 5.11 shows the initial information determined by live-variable analysis for basic block #8 in the bubblesort example. This information indicates that only register r1 is live at the beginning and exit of the basic block. Register r2 is defined within the block, possibly making it live, but since the *out* bitset does not contain r2, the dead-code elimination algorithm knows that there isn't of *use* of r2 after this block, and therefore it is dead. The statement that defines r2 is removed, which causes the only use of register `t47` to disappear so the preceding statement which defines `t47` can also be removed.

The VSSC API provides the ability to *animate* the deletion of an instruction. In this case, the instructions will flash red before they are removed. Other features provided by he VSSC API are described in Section A.4.2 of Appendix A.

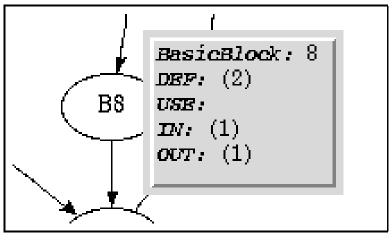

Figure 5.11: Live-variable analysis information for basic block #8 in bubblesort example.

5.4 Example Optimization: Propagating Available Copy Instructions

5.4.1 Copy Propagation

At a point in the program, in which X is a source operand, if there exists a copy instruction (register-to-register copy) [3] $X = Y$ that is currently available, then the operand X at this point in the intermediate code can be replaced with Y since $X = Y$ is available. Determining the *availability* of an instruction is discussed in Section 5.4.2. This valid transformation can cause many copy instructions to become *dead*. They can then be removed by dead-code elimination.

A slightly different version of copy propagation is presented in [30]. In this version, a modified version of available-expression analysis is performed. Instead of

[3] In Simple-SUIF, this instruction is the `CPY_OP` instruction, which copies the value in the `src1` register into the `dest` register.

looking for available expressions, available instruction *forms* are discovered. Given an instruction, $X = Y + Z$, its *instruction form* is that instruction itself. Other occurrences in the intermediate code of that same instruction are occurrences of that instruction form.

Figure 5.12 shows the algorithm for replacing operands with the destinations of available copy instructions. Available-code analysis must be performed first in order to determine which *copy* instructions are available.

```
foreach instruction form i
{
   Bitset K[i] = the set of all instruction
      forms of which i's destination is an operand
}

foreach Basic block B
{
   Bitset currentlyAvailable = B.out
   foreach instruction I in B from last to first
   {
      if there exists an available copy instruction whose dst=I.src1
         I.src1 = dst
      if there exists an available copy instruction whose dst=I.src2
         I.src2 = dst

      if (!currentlyAvailable[I's form])
      {
         currentlyAvailable -= K[I's form]
         if ((I.dst != I.src1) && (I.dst != I.src2))
            currentlyAvailable += I's form
      }
   }
}
```

Figure 5.12: Copy propagation algorithm

5.4.2 Available Code Analysis

Live-variable analysis dealt with the liveness of individual variables. However, not all analysis look solely at individual variables. Available-code analysis (similar to available-expression analysis) looks at statements, which in Simple-SUIF may contain one or more operands and possibly a destination. This analysis is used to determine for a particular point in the intermediate code, which statements are *available* at that point. A statement is available at a point in the intermediate representation if on all paths leading up to this point there is an occurrence of that instruction's *form* and the instruction's operands and destination are not subsequently defined along the paths from that occurrence to the given point.

This analysis algorithm uses bitset data structures to hold availability information. Each basic block contains the bitsets shown in table 5.2. The universal bitset U is also defined to contain all of the unique instruction *forms* in the intermediate code.

Bitset	**Purpose**
in	The set of instruction forms in U available at the entrance of the basic block.
out	The set of instruction forms in U available at the exit of the basic block.
gen	The set of instruction forms in U that occur in B and whose operands are not killed before the exit of B.
kill	The set of instruction forms in U that have an operand that gets defined (killed) in this basic block.

Table 5.2: Bitsets used during available code analysis

Given the *gen* and *kill* bitsets computed for each basic block, the algorithm (a minor variant of available-expression analysis)[1] in Figure 5.13 can be used to compute the *in* and *out* bitsets for each basic block, propagating availability information as far as possible at the same time:

```
InitialB.in = Empty Set
InitialB.out = InitialB.gen

For B != InitialB
   B.out = U - B.kill

change = true
while (change == true)
{
   change = false
   foreach basic block B != InitialB
   {
      B.in = intersection of all P.out's for each predecessor P of B
      oldout = B.out
      B.out = union of B.gen and (B.in - B.kill)
      if (B.out != oldout)
         change = true
   }
}
```

Figure 5.13: Algorithm for available code analysis

Essentially, for each basic block B, statements that are available at the beginning of B are those that are available at the exits of each of the predecessor blocks of B. Those statements that are available at the exit of B are those that are generated in B along with those that are available at the entrance of B and are not killed within B. The above algorithm propagates availability information forward through the flow graph until no new information gets propagated.

5.4.3 Example of Copy Propagation

In this section, a different, shorter example is used instead of the bubblesort used in previous sections. The code in this example, shown in Figure 5.14, is not meant to perform any useful task other than to serve as example code for the following discussion. However, it has many copy instructions that are available.

```
int main()
{
   int i, j, k;

   i = 4;
   j = i;
   if (j == 3) goto L1;
      k = 2;

L1:
   i = j+7;

   while (1)
      i++;
}
```

```
Procedure main:
BASICBLOCK 0:

BASICBLOCK 1:
        ldc  (s.32)     t4 = 4
        cpy  (s.32)     r1 = t4
        cpy  (s.32)     r2 = r1
        ldc  (s.32)     t5 = 3
        seq  (s.32)     t6 = r2, t5
        bfls            t6, L2

BASICBLOCK 2:
        jmp             L1

BASICBLOCK 3:
L2:
        ldc  (s.32)     t7 = 2
        cpy  (s.32)     r3 = t7

BASICBLOCK 4:
L1:
        ldc  (s.32)     t8 = 7
        add  (s.32)     t9 = r2, t8
        cpy  (s.32)     r1 = t9

BASICBLOCK 5:
L6:
        ldc  (s.32)     t10 = 1
        add  (s.32)     t11 = r1, t10
        cpy  (s.32)     r1 = t11
L4:
        jmp             L6

BASICBLOCK 6:
L5:
L3:
        ldc  (s.32)     t12 = 0
        ret             t12
```

Figure 5.14: C and Simple-SUIF versions of copy propagation example

Figure 5.15 shows the bitset information for several basic blocks in the intermediate representation of the simple example program after available-code analysis has been performed. The bits in the bitsets used in this analysis do not refer to registers as they did in live-variable analysis. Instead they refer to instruction forms. Each unique instruction form in the intermediate presentation is assigned a different id, which is the bit used to represent that form in the bitsets. When the node for basic block 0 is clicked on, a mapping appears in the data window that shows the id assigned to each unique instruction form.

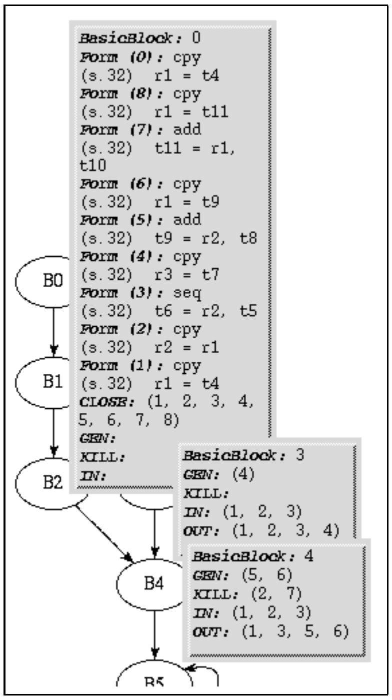

Figure 5.15: Various results after performing available expression analysis

When the copy-propagation algorithm in Figure 5.12 is performed, two instructions end up having one of their sources replaced by the destination of an available copy instruction. Figure 5.16 shows that an instruction in basic block #1 and an instruction in basic block #4 had a source register changed because the copy instruction, `cpy  (s.32)      r1 = t4` was available.

```
BasicBlock 1:
        ldc   (s.32)      t4 = 4
        cpy   (s.32)      r1 = t4
        cpy   (s.32)      r2 = r1
        ldc   (s.32)      t5 = 3
        seq   (s.32)      t6 = r1, t5
        bfls              t6, L2

BasicBlock 4:
L1:
        ldc   (s.32)      t8 = 7
        add   (s.32)      t9 = r1, t8
        cpy   (s.32)      r1 = t9
```

Figure 5.16: Instructions that changed as a result of copy propagation on example in Figure 5.14

5.5 Example: Register Allocation

Once the register-interference graph for a procedure has been constructed, register allocation can be performed using graph coloring. Graph coloring refers to the process of assigning a color to each node in the graph in such a way that no two adjacent nodes have the same color. The number of colors available is equal to the number of physical machine registers and each color maps to an individual machine register. If the graph can be colored, this means that all interfering symbolic registers can be assigned different physical machine registers. In other words, two symbolic registers that have overlapping *live* ranges will not collide, because they will be stored in two different physical machine registers.

The problem of graph coloring is NP-complete. However, several heuristic algorithms exist. One such heuristic is Chaitin's [6] graph coloring heuristic. Assuming that the register spilling is not being performed and the number of physical machine registers is k, this heuristic can be implemented by applying the algorithm shown in Figure 5.17 to an existing register-interference graph.

```
While there are still nodes left in the graph
{
   If there exists a node n with less than k neighbors
   {
      push n onto a stack
      remove n and its edges from the graph
   }
   else
      return (graph cannot be colored using k colors via this heuristic)
}

/* Graph is k-colorable, now assign the colors */

While the stack is not empty
{
   Remove node n from the stack
   Reinsert node n and its edges into the graph
   Assign n a color that is different from those colors of its neighbors
}
```

Figure 5.17: Graph-coloring heuristic algorithm for register-interference graph

For this example, we use the simple program shown in Figure 5.18. Before performing register allocation, we must first construct the register-interface graph, which is explained in Section 5.2.3. Figure 5.19 shows the resulting register-interface graph for this example.

```
int main(int argc, char **argv)
{
   int i, j, k;

   j = 1;
   for (i=0; i<=23; i++)
   {
      j = (k*k)+(i+j);
   }
}
```

```
Procedure main:
BASICBLOCK 0:

BASICBLOCK 1:
        ldc  (s.32)     t6 = 1
        cpy  (s.32)     r4 = t6
        ldc  (s.32)     t7 = 0
        cpy  (s.32)     r3 = t7

BASICBLOCK 2:
L3:
        mul  (s.32)     t8 = r5, r5
        add  (s.32)     t9 = r3, r4
        add  (s.32)     t10 = t8, t9
        cpy  (s.32)     r4 = t10
L1:
        ldc  (s.32)     t11 = 1
        add  (s.32)     t12 = r3, t11
        cpy  (s.32)     r3 = t12
        ldc  (s.32)     t13 = 23
        sl   (s.32)     t14 = t13, r3
        bfls            t14, L3

BASICBLOCK 3:
L2:
        ldc  (s.32)     t15 = 0
        ret             t15
```

Figure 5.18: C and Simple-SUIF versions of register allocation example

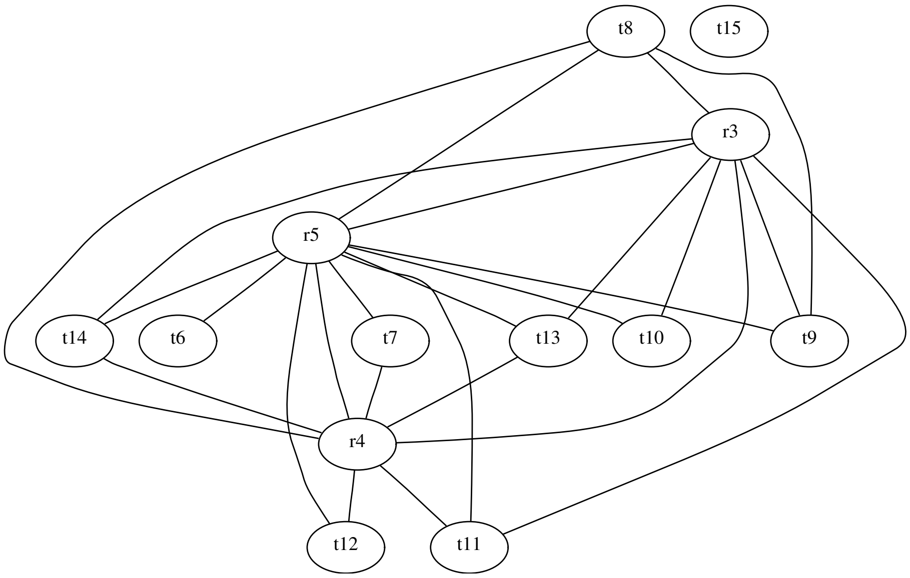

Figure 5.19: Register-interference graph for simple example in Figure 5.18

Assuming that our target architecture contains 32 integer registers, Figure 5.20 shows the intermediate code after the register allocation is performed using the graph-coloring heuristic described in Figure 5.17.

```
Procedure main:
BASICBLOCK 0:

BASICBLOCK 1:
        ldc  (s.32)     m0 = 1
        cpy  (s.32)     m0 = m0
        ldc  (s.32)     m1 = 0
        cpy  (s.32)     m1 = m1

BASICBLOCK 2:
L3:
        mul  (s.32)     m2 = m3, m3
        add  (s.32)     m0 = m1, m0
        add  (s.32)     m0 = m2, m0
        cpy  (s.32)     m0 = m0
L1:
        ldc  (s.32)     m2 = 1
        add  (s.32)     m1 = m1, m2
        cpy  (s.32)     m1 = m1
        ldc  (s.32)     m2 = 23
        sl   (s.32)     m2 = m2, m1
        bfls            m2, L3

BASICBLOCK 3:
L2:
        ldc  (s.32)     m0 = 0
        ret             m0
```

Figure 5.20: Intermediate code of example in Figure 5.18 after register allocation

5.6 Summary

This chapter demonstrated the various type of graphs and transformations that have been implemented with the VSSC framework. The graphs implemented include: a flow graph, which describes the flow-of-control between basic blocks, a directed acyclic graph, which shows commonly computed subexpression, and a register-interference graph, which shows those registers whose live ranges overlapped and thus interfere. The code transformations implemented were: the removal of dead code, copy propagation, and register allocation.

These examples show many of the capabilities provided by the VSSC framework and are a representation of the types of transformations that can be implemented.

Chapter 6

Conclusions and Future Direction

The Visual Simple-SUIF Compiler system is a completely interactive framework that facilitates the study of back-end optimizations. A tightly integrated system of free software components has been combined to produce a tool that can be used to visualize a code transformation step-by-step, perform transformations in any order, and undo a previously executed transformation. The SUIF compiler and Simple-SUIF were used to implemented the intermediate format and the ability to selectively apply transformations to the intermediate code. Tcl/Tk was used as the language to implement the graphical user interface for the VSSC framework. Finally, the `DOT` drawing package was used to allow the GUI to display various types of graph data structures.

A tool such as VSSC contributes greatly to the area of compiler research. The capabilities provided by VSSC aid two groups of individuals in compiler research. First of all, researchers in the area of compiler optimizations can use the VSSC framework to develop and test new code transformations. VSSC allows researchers to analyze how a new transformation modifies the intermediate code and how other transformations can be affected by the changes made by a transformation. Transformations can be applied in any other and undone multiple times.

Students are the second group of individuals that can benefit from using VSSC. VSSC facilitates classroom instruction of compiler optimizations. The ability to step through an optimization allows the student to visualize code optimizations at his or her own pace. The ability to undo a transformation allows the students to repeat the same transformation multiple times without having to restart the compiler. The viability of VSSC as a teaching aid was demonstrated in the graduate compiler course at the University of California Riverside. Students implemented basic block detection, various data-flow analysis, global common-subexpression elimination, and register allocation using VSSC.

6.1 Future Work

Currently, the VSSC framework is a viable solution as a tool for analyzing back-end optimizations in a compiler. However, there are several features that should be implemented to provide greater ease of use and functionality:

- Currently, VSSC allows backstepping at the granularity of transformations and allows the user to step forward through a transformation. The ability to step backwards within the granularity of a transformation should be implemented to provide greater debugging capabilities. Time constraints prevented this feature from being implemented in the current version of the VSSC framework.

- When a code transformation is selected by the user, the optimization is performed and its actions are saved to a log with is then *played* back to the user as they step through the transformation. A nice feature would be the ability to save a transformation log generated by a transformation that has been applied. This log could then be reloaded at a later time and played back as though the transformation had just been applied. Instructors in compiler courses can create such logs to give to their students to strengthen classroom instruction.

- Currently, live-variable analysis, dead-code elimination, available-code analysis, copy propagation, and a simple form of register allocation have been implemented by the author. There are numerous other algorithms and code transformation techniques that could be implemented to further demonstrate the capabilities of the VSSC framework.

- The VSSC framework is currently unable to perform analysis above the level of a procedure due to limitations in the Simple-SUIF library. Perhaps a workaround could be developed to implement the ability to perform inter-procedural analysis and the construction of call graphs.

Finally, constructive feedback from those who have used the VSSC framework can be used to further improve its functionality.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.

[2] Kristy Andrews, Robert R. Henry, and Wayne K. Yamamoto. Design and implementation of the UW illustrated compiler. Technical Report 88-03-07, University of Washington, March 1988.

[3] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography, June 1994.

[4] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, Georgia, USA, June 1988.

[5] Mickey R. Boyd and David B. Whalley. Graphical visualization of compiler optimizations. *Journal of Programming Languages*, pages 69–94, June 1995.

[6] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 98–105. ACM, ACM, 1982. Available as SIGPLAN Notices 17(6) June 1982.

[7] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[8] Christopher W. Fraser and David R. Hanson. A code generation interface for ANSI C. Technical Report TR-270-90, Department of Computer Science, Princeton University, June 1990.

[9] Christopher W. Fraser and David R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[10] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.

[11] Christopher W. Fraser, David R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[12] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG–fast optimal instruction selection and tree parsing. Technical Report TR 1066, Computer Sciences Department, University of Wisconsin-Madison, December 1991.

[13] Christopher W. Fraser and Alan L. Wendt. Integrating code generation and optimization. In *SIGPLAN '86 Symposium on Compiler Construction*, pages 242–248, Palo Alto, CA, June 1986. Association for Computing Machinery, SIGPLAN.

[14] Christopher W. Fraser and Alan L. Wendt. Automatic generation of fast optimizing code generators. *SIGPLAN Notices*, 23(7):79–84, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[15] Carsten Friedrich. The ffgraph library. Technical Report MIP-9520, Fakultät für Mathematik und Informatik, Universität Passau, December 1995.

[16] M. Fröhlich and M. Werner. The graph visualization system daVinci - A user interface for applications. Technical Report 5/94, Department of Computer Science; University of Bremen, September 1994.

[17] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[18] James Gosling and Henry McGilton. The java language environment – A whitepaper. Technical report, Sum Microsystems, October 1995.

[19] Michael Himsolt. *Graph^{Ed} User Manual*, 1990.

[20] S. C. Johnson. YACC: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[21] E. Koutsofios and S. C. North. Drawing graphs with *dot.* Technical report, AT&T Bell Laboratories, Murray Hill, NJ, October 1993.

[22] E. Koutsofios and S. C. North. TCLDOT(1). *Unix Manual Page*, March 1995.

[23] M. E. Lesk and E. Schmidt. Lex – A lexical analyzer generator. Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975.

[24] Jing Yee Lim. A visual browser for SUIF. In *SUIF Compiler Workshop*, January 1996.

[25] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Note STAN//CS-TN-95-22, Stanford University, Department of Computer Science, August 1995.

[26] S. Naher. LEDA — a library of efficient data types and algorithms. *Lecture Notes in Computer Science*, 665:710–??, 1993.

[27] Stephen C. North and Eleftherios Koutsofios. Application of graph visualization. In *Proceedings of Graphics Interface '94*, pages 235–245, Banff, Alberta, Canada, May 1994. Canadian Information Processing Society.

[28] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, Reading, Massachusetts, 1994.

[29] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[30] Thomas H. Payne. Compiler design. CS201 Lecture Notes, October 1996.

[31] Norman Ramsey. Literate-Programming can be simple and extensible. Technical report, Department of Computer Science, Princeton University, Princeton, New Jersey, October 1993.

[32] G. Sander. VCG visualization of compiler graphs. *User documentation V1.130*, February 1995.

[33] G. Sander. Graph layout for applications in compiler construction. Technical Report A/01/96, Universität des Saarlandes, February 1996.

[34] Richard Stallman. *Using and porting GNU CC.* Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, last updated 26 november 1995 for version 2.7.2 edition, 1995.

[35] Richard Stallman and Roland H. Pesch. *Debugging with GDB: the GNU source-level debugger.* 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 4.12, for GDB version 4.14 edition, January 1995.

[36] Stanford Compiler Group. *The Simple-SUIF Compiler Guide*, 1.0 edition. A simple interface to SUIF for compiler courses.

[37] Stanford Compiler Group. *The SUIF Library*, 1.0 edition. A set of core routines for manipulating SUIF data structures.

[38] Steven Tjiang. Twig reference manual. Comp. Sci. Tech. Rep. 120, AT&T Bell Laboratories, January 1986.

[39] Steven W. K. Tjiang and John L. Hennessy. Sharlit—A tool for building optimizers. In *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, July 1992.

[40] Brent Welch. *Practical Programming in Tcl and Tk.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1995.

[41] Chris Wilson. Announcing the new SUIF visual browser package. *suif-announcesuif.stanford.edu mailing list*, April 1996.

[42] 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. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[43] 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. *An Overview of the SUIF Compiler System.*

[44] Michael Wolfe. dflo compiler graph display tool. *USENET posting to comp.compilers*, August 1996.

Appendix A

VSSC User Manual

A.1 Introduction

This document is provided as a user's manual for using the Visual Simple-SUIF package (VSSC). A user should be able to effectively use the VSSC package to develop graphical compiler optimizations by just reading this document and not reading the thesis describing VSSC. In addition to the more technically oriented information describing how to get VSSC compiled and installed at your site and tips to make writing compiler optimizations with VSSC easier, the information contained within this document is a subset of the information contained within the thesis that this document is an appendix to.

This document begins by describing the SUIF and Simple-SUIF packages. The VSSC package is then covered in detail. Examples are provided throughout the document.

A.2 SUIF

The SUIF compiler system, developed by a team of researchers at Stanford University, is centered around the design of its intermediate format, SUIF (Stanford University Intermediate Format). The system has been designed and organized in such a way that it is easy to modify and extend for your own personal needs. Because of its flexibility, many researchers around the world use the SUIF compiler system to evaluate new compiler techniques and perform research in the area of compilers. While the SUIF compiler may not be the fastest or most robust. Its flexibility and extensibility outweighs these possible shortcomings for most people.

The SUIF system is organized into two parts. The first part is a *core* which manages the intermediate format. The second part is a set of compiler passes which perform some transformations on the intermediate format. Usually, each pass reads in the intermediate code, performs some transformation, analysis, or optimization, and then writes out the intermediate code. Since each pass usually exists as a separate executable in the SUIF system, passes can be run in any order in the compilation process. To aid in the creation of SUIF passes, the SUIF system contains a robust set of libraries and support routines.

A.3 Simple-SUIF

Due to the fact that SUIF compiler is a complete ANSI C compiler, it is very complete and complex. Because it is a little too complex for use in a college course in compilers, the SUIF group at Stanford developed a package called **Simple-SUIF** which acts as a *wrapper* for SUIF by providing a simplified interface to the SUIF compiler and the intermediate format generated by the SUIF compiler. This simplified interface allows students to write their own optimizations for a fully-functional ANSI C compiler.

Information about Simple-SUIF can be found in the document, "The Simple-SUIF Compiler Guide"[1] for version 1.0 of Simple-SUIF. Most of the information presented in this section comes from that document. More detailed explanations and descriptions of the topics mentioned in this section can be found in that document.

VSSC is built around Simple-SUIF. However, its interaction with Simple-SUIF is slightly different. Because of this, the interaction with Simple-SUIF described in chapter 2 of the Simple-SUIF documentation should be ignored. Section A.4 of this document describes how to use VSSC with Simple-SUIF and how its interaction is slightly different.

A.3.1 Simple-SUIF Intermediate Format

The instructions in Simple-SUIF's intermediate format resemble assembly language instructions (*op dst,src1,src2*) or three-address C instructions (*dst = src1 op src2*). Each instruction has an unique opcode associated with it. The instructions are grouped into six different categories called *instruction formats*. Table A.1 shows all the valid Simple-SUIF instructions. For each instruction, the following information is also shown: its opcode, its Simple-SUIF name, its instruction format, and a short explanation of that instruction.

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

Simple-SUIF Instructions			
Opcode	**Instr. Name**	**Instruction Format**	**Purpose**
No operand instructions			
NOP_OP	nop	BASE_FORM	No nothing at all
One source operand (src1) instructions			
RET_OP	ret	BASE_FORM	Return from a procedure
Two source operand (src1, src2) instructions			
STR_OP	str	BASE_FORM	Store the value in the **src2** register at the address contained in the **src1** register
MCPY_OP	mcpy	BASE_FORM	Memory-to-memory copy
Unary instructions (dst, src1)			
CPY_OP	cpy	BASE_FORM	Copy the **src1** register to the **dst** register
CVT_OP	cvt	BASE_FORM	Convert the **src1** register to the result type and put it in the **dst** register
NEG_OP	neg	BASE_FORM	Negation
NOT_OP	not	BASE_FORM	Bit-wise inversion
LOAD_OP	load	BASE_FORM	Load the value at the address contained in the **src1** register and put it in the **dst** register
Binary instructions (dst, src1, src2)			
ADD_OP	add	BASE_FORM	$\mathbf{dst} = \mathbf{src1} + \mathbf{src2}$
SUB_OP	sub	BASE_FORM	$\mathbf{dst} = \mathbf{src1} - \mathbf{src2}$
MUL_OP	mul	BASE_FORM	$\mathbf{dst} = \mathbf{src1} * \mathbf{src2}$
DIV_OP	div	BASE_FORM	$\mathbf{dst} = \mathbf{src1}/\mathbf{src2}$
REM_OP	rem	BASE_FORM	$\mathbf{dst} = \mathbf{src1}\%\mathbf{src2}$
MOD_OP	mod	BASE_FORM	$\mathbf{dst} = abs(\mathbf{src1}\%\mathbf{src2})$
AND_OP	and	BASE_FORM	Bit-wise AND
IOR_OP	ior	BASE_FORM	Bit-wise inclusive OR
XOR_OP	xor	BASE_FORM	Bit-wise exclusive OR
ASR_OP	asr	BASE_FORM	Signed shift right
LSL_OP	lsr	BASE_FORM	Unsigned shift right
LSR_OP	lsl	BASE_FORM	Unsigned shift left
ROT_OP	rot	BASE_FORM	Rotate value in **src1** register left (positive value) or right (negative value) by the amount specified in the **src2** register
SEQ_OP	seq	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} == \mathbf{src2})$
SNE_OP	sne	BASE_FORM	$\mathbf{dst} = (\mathbf{src1}! = \mathbf{src2})$
SL_OP	sl	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} < \mathbf{src2})$
SLE_OP	sle	BASE_FORM	$\mathbf{dst} = (\mathbf{src1} <= \mathbf{src2})$
Branch and jump instructions			
JMP_OP	jmp	BJ_FORM	Unconditional jump: **goto target**
BTRUE_OP	btru	BJ_FORM	Branch if true: **if (src1) goto target**
BFALSE_OP	bfls	BJ_FORM	Branch if false: **if (!src1) goto target**
Miscellaneous			
LDC_OP	ldc	LDC_FORM	Load a constant value
CALL_OP	call	CALL_FORM	Call a procedure
MBR_OP	mbr	MBR_FORM	Multi-way branch
LABEL_OP	lab	LABEL_FORM	Label pseudo-instruction

Table A.1: Valid Simple-SUIF instructions

A Simple-SUIF instruction is represented by the **simple_instr** structure. Figure A.1 shows the **simple_instr** structure and its contents. This figure does not show the format of the structures used within the **simple_instr** structure. More information about the contents of the **simple_reg**, **simple_sym**, **simple_immed**, and **simple_type** structures can be found in the **simple.h** header file.

```
simple_op opcode;          /* the opcode */
simple_type *type;         /* type of the result */
struct simple_instr *next; /* ptr to next instruction */
struct simple_instr *prev; /* ptr to previous instruction */

union u;    /* the variant part of the union is determined
               by the result of simple_op_format(opcode) */

  /* BASE_FORM */
  struct base {
     simple_reg *dst;  /* destination */
     simple_reg *src1; /* source 1 */
     simple_reg *src2; /* source 2 */
  }

  /* BJ_FORM */
  struct bj {
     simple_sym *target; /* branch target label */
     simple_reg *src;    /* source register */
  }

  /* LDC_FORM */
  struct ldc {
     simple_reg *dst;    /* destination */
     simple_immed value; /* immediate constant */
  }

  /* CALL_FORM */
  struct call {
     simple_reg *dst;   /* return value destination */
     simple_reg *proc;  /* address of the callee */
     unsigned nargs;    /* number of arguments. */
     simple_reg **argsl /* array of arguments */
  }

  /* MBR_FORM */
  struct MBR {
     simple_reg *src;      /* branch selector */
     int offset;           /* branch selector offset */
     simple_sym *deflab;   /* label of default target */
     unsigned ntargets;    /* number of possible targets */
     simple_sym **targets; /* array of labels */
  }

  /* LABEL_FORM */
  struct label {
     simple_sym *lab; /* the symbol for this label */
  }
```

Figure A.1: **simple_instr** structure used to represent a Simple-SUIF instruction

As you can see from the figure, the **simple_instr** structure contains a member **u** which is a union of many other structures. A union is used to save memory since an instruction can only be of type of instruction format. Each of the structures in the union represents a different instruction format. Given a Simple-SUIF instruction it is very easy to access the data contained within the union. For example, say you wanted to print out all the branch target labels that existed in a linked list of Simple-SUIF instructions. Those instructions which would have branch targets are those that belong to the **BJ_FORM** and **MBR_FORM** (branch and multi-way branch) formats. The code in Figure A.2 demonstrates how to accomplish this task (the call to **simple_op_format** is described in section A.3.2):

```
void print_branch_targets(simple_instr *inlist)
{
   simple_instr *curr_instr;

   curr_instr = inlist;

   /* foreach instruction in the linked list of Simple-SUIF instructions. */

   while (curr_instr != NULL)
   {
      /* See if this instruction belongs to the BJ_FORM or MBR_FORM
         instruction formats. */

        switch (simple_op_format(curr_instr->opcode))
        {
           case BJ_FORM:
              /* This instruction format has 1 target. */
              printf("Branch target: %s\n", curr_instr->u.bj.target->name);
              break;
           case MBR_FORM:
              /* This instruction format has mulitple targets
                 including a default target (like a C switch
                 statement) */
              printf("MBR: default label: %s\n", curr_instr->u.mbr.deflab->name);
              for (int i=0; i<(int)curr_instr->u.mbr.ntargets; i++)
              {
                 printf("MBR: target #%i: %s\n",
                         i, curr_instr->u.mbr.targets[i]->name);
              }
              break;
           default:
              /* Ignore everyone else. */
              break;
        };
        curr_instr = curr_instr->next;
   }
}
```

Figure A.2: Example demonstrating the different instruction formats

A.3.2 Simple-SUIF API

The application program interface (API) of a library is the documented set of commands that are available to the user of that library. Simple-SUIF provides a small API which allows you to easily work with the Simple-SUIF library. The API contains the commands shown in Figure A.3.

simple_instr *new_instr(simple_op op, simple_type *t) Allocate and initialize a new Simple-SUIF instruction. It only sets the opcode, format, and return type fields. You need to setup everything else (**dst**, **src1**, and **src2**) as needed.

void free_instr(simple_instr *s) Deallocates a Simple-SUIF instruction.

simple_reg *new_register(simple_type *t, reg_kind k) Allocates a new Simple-SUIF register.

simple_sym *new_label() Creates a new Simple-SUIF label. This is does not create a new Simple-SUIF instruction (with a format type of LABEL_FORM) but instead inserts this new symbol into the symbol table. You must then call the **new_instr** command above to create the label instruction.

simple_type *get_ptr_type(simple_type *t) Get a type that is a pointer to another type.

char *simple_op_name(simple_op o) Returns the name of an opcode as a text string.

simple_format simple_op_format(simple_op o) Given an opcode, returns the format of that opcode.

Figure A.3: Simple-SUIF API

Simple-SUIF also defines `TRUE` and `FALSE` which can be used in conditional statements.

The following code segment as well as the previous example demonstrate some of the commands in the Simple-SUIF API. In the following example, we have the following linked list of instructions with the variable **curr_instr** pointing the last one:

```
cpy  (s.32)     r3 = t9
ldc  (s.32)     t68 = 1     <- current_instr
```

Say we want to add a new instruction, **add (s.32) t69 = r3, t68**, right after **curr_instr**. The code to do this is shown in Figure A.4.

```
simple_instr *ni;

ni = new_instr(ADD_OP, simple_type_signed);

/* Insert new instruction into current linked list of instructions
   right after curr_instr. */

ni->next = curr_instr->next;
ni->prev = curr_instr;
curr_instr->next = ni;
ni->next->prev = ni;

/* Fill in the blanks in the new instruction. */

/* dst. Create a new temporary register of type signed int. */

ni->u.base.dst = new_register(simple_type_signed, TEMP_REG);

/* For src1 and src2, simple use the same pointers the two previous
   instructions use. */

ni->u.base.src1 = curr_instr->prev->u.base.dst;
ni->u.base.src2 = curr_instr->u.ldc.dst;

/* NOTE: If you create an instruction in which a src field is empty
   (ie src2 is empty in a cpy instruction), you need to set that
   empty field equal to NO_REGISTER */
```

Figure A.4: Example C code to add a new Simple-SUIF instruction

As you can see in the example above, the return type of the newly created instruction is a signed integer. **simple_type_signed** is global variable used in several places in the above code above to represent that return type. It is among several "base types" Simple-SUIF defines which are shown in Figure A.5.

VOID_TYPE Used to indicate that there is no value present.

SIGNED_TYPE Signed integers.

UNSIGNED_TYPE Unsigned integers.

FLOAT_TYPE Floating-point values.

ADDRESS_TYPE Pointers.

RECORD_TYPE Structures, unions, and arrays.

Figure A.5: Simple-SUIF base types

The above example also makes reference to an enumerated type **TEMP_REG**. **TEMP_REG** is one of three types of registers used in Simple-SUIF. The three types are **TEMP_REG**, **PSEUDO_REG**, and **MACHINE_REG**. You can find more information about these three types of registers in the Simple-SUIF documentation.

A.3.3 Example of Simple-SUIF

We conclude this section on Simple-SUIF with a side-by-side comparison of a sample C file and it Simple-SUIF equivalent. These are shown in Figure A.6. You should be able to associate the Simple-SUIF instructions with their corresponding C instructions.

```
test()
{
   int a, b, c, d, e, f, g;

   a = 1;
   b = 2;

   c = a + b;
   g = f - e;
   d = d - c;

   while (a > 0) {
      a = c - d;
      if (g == d * e)
         b = b - 1;
      c = c * f;
   }

   f = g - a;
   c = b + e;
}
```

```
Procedure test:
        ldc  (s.32)     t8 = 1
        cpy  (s.32)     r1 = t8
        ldc  (s.32)     t9 = 2
        cpy  (s.32)     r2 = t9
        add  (s.32)     t10 = r1, r2
        cpy  (s.32)     r3 = t10
        sub  (s.32)     t11 = r6, r5
        cpy  (s.32)     r7 = t11
        sub  (s.32)     t12 = r4, r3
        cpy  (s.32)     r4 = t12
        ldc  (s.32)     t13 = 0
        sl   (s.32)     t14 = t13, r1
        bfls            t14, L1
L4:
        sub  (s.32)     t15 = r3, r4
        cpy  (s.32)     r1 = t15
        mul  (s.32)     t16 = r4, r5
        seq  (s.32)     t17 = r7, t16
        bfls            t17, L5
        ldc  (s.32)     t18 = 1
        sub  (s.32)     t19 = r2, t18
        cpy  (s.32)     r2 = t19
L5:
        mul  (s.32)     t20 = r3, r6
        cpy  (s.32)     r3 = t20
L2:
        ldc  (s.32)     t21 = 0
        sl   (s.32)     t22 = t21, r1
        btru            t22, L4
L3:
L1:
        sub  (s.32)     t23 = r7, r1
        cpy  (s.32)     r6 = t23
        add  (s.32)     t24 = r2, r5
        cpy  (s.32)     r3 = t24
        ldc  (s.32)     t25 = 0
        ret             t25
```

Figure A.6: Side-by-side comparison of C and Simple-SUIF

A.4 VSSC

A.4.1 Introduction

Most modern compilers today perform the standard code optimizations described in compiler textbooks. SUIF's flexibility allows compiler researchers to develop new optimization routines with it easier than with other more complex compilers such as GCC. While a new optimization routine may look good on paper, sometimes its effectiveness and the impact it can make on intermediate code isn't apparent unless the researcher can visualize the transformations made by the optimization. Being able to *step* through the transformations also yields benefits. Anyone who has ever used a debugger such a **gdb** knows that one of the best features of a debugger to the ability to *step* through the code. *Stepping* through code allows the user to progress at his/her own pace and view the current *state* of the program at any point.

VSSC provides the same benefits as a debugger when dealing with code optimizations. When stepping through the transformations made by an optimization, a VSSC user can view the current state of the intermediate code, a graph representing the flow between basic blocks or perhaps a directed acyclic graph showing dependencies between operands in instructions, and the current bitsets for each basic block when doing dataflow analysis. A side benefit of the VSSC system is that it can also be used in compiler courses at academic institutions as a teaching tool. By stepping through the implementations of the classic optimization techniques such as dead code elimination, copy propagation, register allocation, etc., students can gain a better understanding of these technique because they will be viewing them as they happen graphically.

VSSC is based upon Simple-SUIF, however it does not use Simple-SUIF in the way it was originally intended. Simple-SUIF was designed so that a program linked with the Simple-SUIF library could only make one optimization pass for each procedure in an input file during the execution of a compiler optimization. In other words, say you wrote a program, linked with the Simple-SUIF library, that performs dead code elimination. All that program would do is read in the SUIF file, perform that optimization, and write the new SUIF code back out to a file. Simple-SUIF provides no mechanism for performing multiple passes (ie optimizations) during a single execution. While this follows along nicely with the rest of the design philosophy for SUIF, that all code transformations are performed as a series of "passes", it does not allow one to construct a single Simple-SUIF "optimizing compiler" that can perform multiple optizations during a single execution of the "compiler".

A VSSC compiler overcomes this shortcoming of Simple-SUIF. It does so by managing all the SUIF and Simple-SUIF libraries and SUIF executables into a single executable that allows the user to be able to perform multiple optimizations during the execution of the "visual" VSSC compiler.

Figure A.7 shows what the VSSC compiler looks like while it is running. The screen consists of three main areas of which you have control over.

The first area (Current Status) in the top left simply acts as a general information area. Since VSSC allows the user to step through the optimization (much like a *step* in a debugger), this first area contains two buttons that allow the user to be able to *step* forward and backward. During each step, any number of actions (ie adding/deleting instructions/graph nodes) can occur. It is up to the optimization writer to decide what happens.

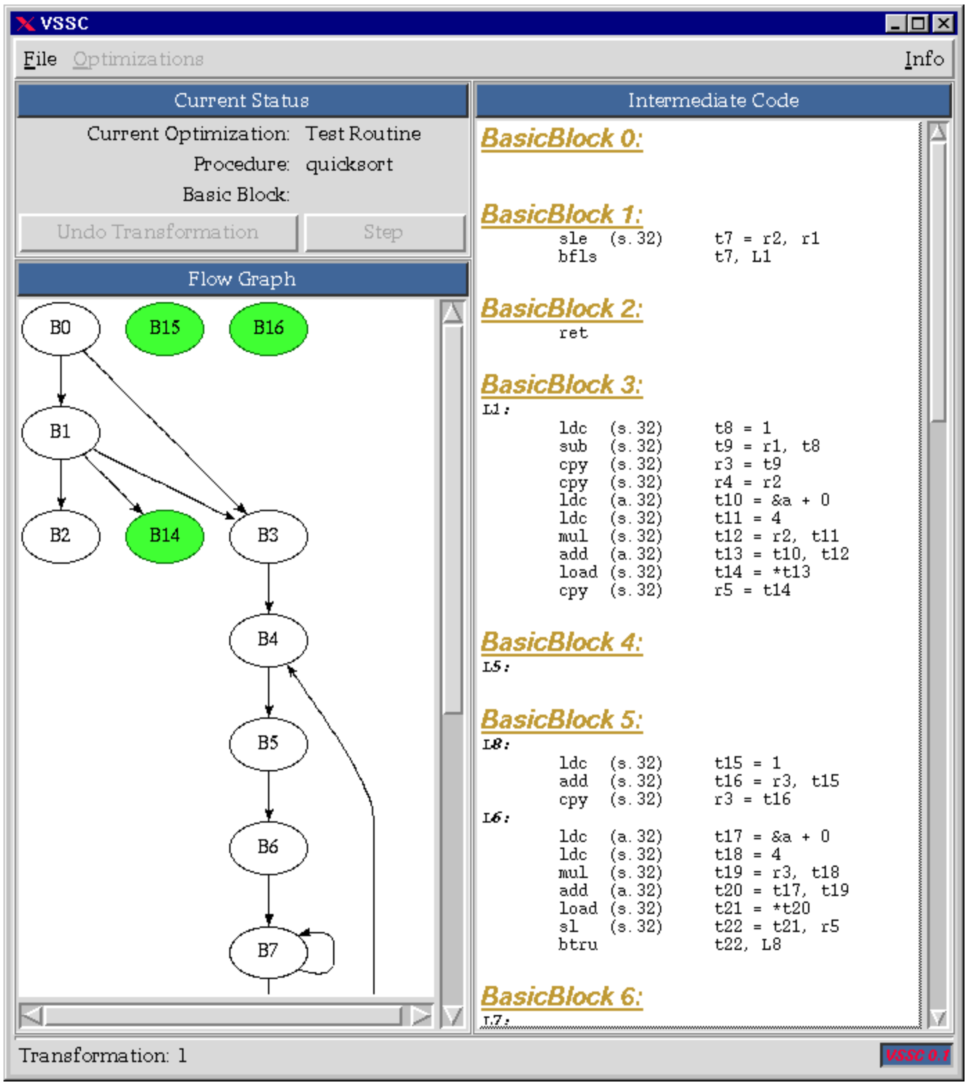

Figure A.7: Screenshot of VSSC compiler in action

The area below the first area contains a graph widget. In this area the optimization writer can create graphs. The most common types of graphs that could be created are flowgraphs and directed acyclic graphs (DAGs). Each node in the graph widget can have arbitrary data associated with it. This data is displayed when the user clicks on the graph node with the left mouse button. Clicking in the box that contains the data hides it. The last area on the right hand side of the screen is a text widget which contains the Simple-SUIF intermediate code. The code presented in this widget usually is contained within basic blocks (as they are in the figure).

The next section describes the API available to you that allows you, the optimization writer, to interact with these areas on the screen.

A.4.2 VSSC API

The VSSC API is grouped into several sections described below. An example is provided for most of the API routines. More examples can be found in section A.4.6 and A.5.6. To use the VSSC API, `#include <vssc_simple.h>` at the top of your files. The GUI and GRAPH variables used below are global variables. You do not need to `extern` them.

Optimization Routines

This section describes those commands for dealing with the optimization routines that you write. Figure A.8 shows these commands. Section A.4.6 describes how and when you need to use these routines and shows examples of their use.

void vssc_register_opt(char *name, char *flagname, simple_instr *(*procedure)(simple_instr *, char *))
: The first parameter to **vssc_register_opt** is a full text string giving the name of the optimization. This is the name which appears as an entry in a VSSC compiler's "optimizations" menu. The second parameter is a text string which is the flag used to signify this optimization when running a VSSC compiler in non-graphical mode (see Section A.5.3 for more details on this mode). The last parameter is the procedure (whose *signature* must match that of the parameter above) to call to perform the optimization.

GUI→step() As mentioned earlier in this document, VSSC allows the user to *step* through the optimization. The optimization writer decides what should occur during each *step*. A *step* boundary is specified by calling `GUI→step()`. When this command is executed, the VSSC API commands that were called since the last call to `GUI→step()` would be executed when the user presses the **Step** button.

vssc_init_suif(int, char **) After all optimization routines have been registered, a call to this sets everything in motion. You shouldn't have any code after this call. The first parameter is usually `argc` and the second parameter is usually `argv`. See Section A.4.6 to see an example of its use. Usually, your **main.cc** contains only calls to **vssc_register_opt** to register the various optimizations routines you have written for your VSSC compiler followed by a call to **vssc_init_suif** to get the VSSC system started.

Figure A.8: Optimizations API

Current Status Area

This section describes those commands for modifying what is displayed in the Current Status area. Figure A.9 shows these commands.

Intermediate Code (Text Widget)

This section describes those commands which control what basicblocks and instructions are displayed in the text widget. VSSC defines a BasicBlock class for you called `BasicBlock_base`. VSSC expects you to use this class when using Basic Blocks in VSSC. Figure A.10 shows the header file for the BasicBlock_base class:

void GUI→set_procedure_name(char *string); Sets the string displayed in the **Procedure Name:** field.

void GUI→set_basicblock_num(int number); Set the number displayed in the **BasicBlock:** field. Use **BLANK** for the parameter if you don't want any value displayed.

```
GUI->set_basicblock_num(3);
```

Figure A.9: Current Status Area API

```
#ifndef BASICBLOCK_BASE_H
#define BASICBLOCK_BASE_H

extern "C"
{
#include <simple.h>
}

#include <suif.h>

class BasicBlock_base
{
 public:
   BasicBlock_base(simple_instr *start_instr,
                   simple_instr *end_instr,
                   unsigned int new_id);
   unsigned int size() {return count;}
   unsigned int get_id() {return id;}
   simple_instr *get_start() {return start;}
   simple_instr *get_end() {return end;}

   void print();

 protected:
   unsigned int count;
   unsigned int id;
   simple_instr *start;
   simple_instr *end;
};

#endif
```

Figure A.10: Header file for BasicBlock class

As you can see, this class is very small, simple, and its interface is self-explanatory. You will most likely need to derive your own BasicBlock class in your own code using this class as a parent class. For example, when doing dataflow analysis, your derived BasicBlock class can contain the various bitsets needed in that analysis. See the supplied example code described in section A.4.6 for more details on how to make this derived class. One important thing to keep in mind is that if your derived class overrides any of the above routines in the parent class, those routines need to call the corresponding routine in the parent class. This is most apparent when dealing with the constructor for the BasicBlock_base class. For example, if your derived BasicBlock class overrides the constructor, it also needs to call the constructor in the parent class. The supplied example code described in section A.4.6 demonstrates this.

Figure A.11 shows the API for dealing with the text widget on the screen.

Graph Widget

This section covers those commands which control the information displayed in the graph widget. The graph widget API is shown in Figures A.12 and A.13. VSSC defines a simple Graph class which is sufficient enough but can be used as a base class for your own derived Graph class (like the BasicBlock class in the previous section). You'll probably won't need to derive your own Graph class but if you do decide to, make sure that you follow the same precautions mentioned previously and for any procedure you override, make sure it calls the same procedure in this base class.

A reminder that the GRAPH variable mentioned below is an instantiation of the VSSC Graph class and exists as a global variable (no need to **extern**). Of course, if you derived your own Graph class, you would need to instantiate it as a global variable and the calls to the Graph widget API would change to: `<your graph class>->routine(...);`.

void GUI→add_basicblock(BasicBlock_base *BB, BasicBlock_base *after, bool animate=FALSE); Given a pointer to an instance of the BasicBlock_base class (or a derived class), insert that basic block (along with its instructions automatically) into the text widget. If *after* is NULL, inserts it at the end of the text widget. If *after* is not NULL, it inserts the basic block after the basic block pointed at by *after* in the text widget (a runtime error is generated if the *after* basic block doesn't exist in the text widget yet. *animate* is an optional boolean parameter that specifies whether to animate the insertion of all the instructions.

```
GUI->add_basicblock(BB, NULL, FALSE);
```

void GUI→add_instruction(simple_instr *simple, BasicBlock_base *BB, bool animate=FALSE); Inserts the Simple-SUIF instruction *simple* at the beginning of basic block *BB* in the text widget. A runtime error is generated if *BB* doesn't exist in the text widget yet. *animate* is an optional boolean parameter that specifies whether the insertion of the instruction should be animated.

```
GUI->add_instruction(new_instr, BasicBlocks[3], TRUE);
```

void GUI→add_instruction(simple_instr *simple, simple_instr *after, bool animate=FALSE); Inserts the Simple-SUIF instruction *simple* after the instruction *after* in the text widget. A runtime error is generated if *after* doesn't exist in the text widget yet. *animate* is an optional boolean parameter that specifies whether the insertion of the instruction should be animated.

```
GUI->add_instruction(new_instr, new_instr->prev, TRUE);
```

void GUI→remove_instruction(simple_instr *simple, bool animate=FALSE); Removes the instruction *simple* from the text widget. A runtime error is generated if *simple* doesn't exist in the text widget yet. *animate* is an optional boolean parameter that specifies whether the deletion of the instruction should be animate.

```
GUI->remove_instruction(dead_instr);
```

Figure A.11: Intermediate Code API

void GUI→set_graph_type(char *string); Sets the string displayed at the top of the Graph widget. This value starts out as `Graph`.

```
GRAPH->set_graph_type("Flow Graph");
```

GraphNode* GRAPH→addNode(char *name, bool animate=FALSE) Create a new node in the Graph widget with the name *name*. *animate* is an optional argument which indicates whether the addition of this node should be animated or not. This routines returns a pointer to a `GraphNode` structure which is used by VSSC to represent this node. All further interactions which this node requires the returned `GraphNode` pointer as a parameter. To make things easy, your derived BasicBlock_base class could contain a `GraphNode *` data item that you can save this pointer in (since you usually associate a node in a Flow Graph with a basic block). An error is generated is you try to create a new node with the same `name` as an existing one.

```
GraphNode *GN = GRAPH->addNode("B2", TRUE);
```

void GRAPH→addEdge(GraphNode *, GraphNode *); Given pointers to two VSSC graph nodes, create an edge between them in the graph. You can create a self-edge. An error is generated if either of the graph nodes don't exist or an edge already exists between these nodes.

```
GRAPH->addEdge(GN1, GN2);
```

int GRAPH→nodeExists(char *name); Returns TRUE if the node with the same *name* exists in the graph already and FALSE if it doesn't.

int GRAPH→nodeExists(GraphNode *); Returns TRUE if this node exists in the graph already and FALSE if it doesn't.

int GRAPH→edgeExists(GraphNode *, GraphNode *); Returns TRUE if an edge exists between the two nodes in the graph already and FALSE if it doesn't.

void GRAPH→removeNode(char *name, bool animate=FALSE) Remove a node from the Graph widget. *animate* is an optional argument which indicates whether the deletion of this node should be animated or not. When a node is removed, all edges between that node and other nodes are also removed. An error is generated is you try to remove a node that doesn't exist in this graph.

```
GRAPH->removeNode(GN, TRUE);
```

Figure A.12: Graph widget API

void GRAPH→removeEdge(GraphNode *, GraphNode *); Given pointers to two VSSC graph nodes, removes the edge between them in the graph. An error is generated if the edge doesn't exist already between these nodes.

```
GRAPH->removeEdge(GN3, GN7);
```

void GRAPH→addDataItem(GraphNode *, char *key, char *data); Associates arbitrary textual data with a graph node. Data is supplied as a *key* along with actual *data*. When displayed (when the user clicks on the graph node with the left mouse button), *key* is italicized following by *data* as shown in Figure A.14. Data items for a graph node are overwritten by just calling this routine again with the same *key*. An error is generated if the graph node doesn't exist in this graph.

```
GRAPH->addDataItem(GN2, "live", "(4) (5) (8)");
```

void GRAPH→removeDataItem(GraphNode *, char *key); Removes the *key* data item for the specified node. An error is generated is the graph node doesn't already exist in the graph.

```
GRAPH->removeDataItem(GN6, "live");
```

Figure A.13: Graph widget API

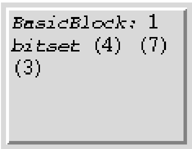

Figure A.14: Box that pops up when user clicks on graph node with left mouse button

Miscellaneous Commands

This section describes extra commands provided in the VSSC library. Figure A.15 shows these commands.

void simple_instr_print(FILE *fd, simple_instr *s) Given a Simple-SUIF instruction, prints a textual representation of it to the supplied file pointer. The file pointer can be for a file (created by a call to **fopen**) or it can be **stdout** or **stderr**. Most of this code comes from the Simple-SUIF **printsimple** command.

char *vssc_simple_text(simple_instr *s) Given a Simple-SUIF instruction, returns a string that represents a textual version of the instruction (as it would appear in **printsimple** or in the text widget.

Figure A.15: Miscellaneous VSSC commands

A.4.3 Installing VSSC Components

Table A.2 lists packages that are required to be installed before VSSC can be used.

GNU C/C++	`ftp://prep.ai.mit.edu/pub/gnu/gcc-2.7.2.tar.gz`
SUIF	`http://suif.stanford.edu/` `ftp://suif.stanford.edu/pub/suif/basesuif-1.1.2.tar.gz`
Simple-SUIF	`ftp://suif.stanford.edu/pub/suif/simplesuif-1.0.0.beta.1.tar.gz`
Tcl/Tk	`http://www.sunlabs.com/research/tcl/` `ftp://ftp.sunlabs.com/pub/tcl/`
DOT/tcldot	`http://www.research.att.com/sw/tools/reuse/`

Table A.2: VSSC components and where to find them

Information on how to install these packages can be found in the documentation included with each package.

A.4.4 Environment Variables

In order to use the VSSC package, a VSSC compiler, or compile a VSSC compiler, the same environment variables that are needed for using SUIF need to be set. The settings shown in Figure A.16 (for either [t]csh or bash) are for the Linux machines in the Department of Computer Science at the University of California Riverside. Insert these commands at the **end** of your shell's startup file (`.[t]cshrc` for [t]csh or `.profile` for bash) so you don't have to type them in each time you log in.

[t]csh

```
setenv MACHINE i586-linux
setenv SUIFHOME /usr/local/suif
setenv SUIFPATH $SUIFHOME/$MACHINE/bin:/usr/bin:/usr/local/bin
setenv COMPILER_NAME gcc
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:$SUIFHOME/$MACHINE/solib
setenv PATH ${PATH}:$SUIFHOME/$MACHINE/bin
```

bash

```
export MACHINE=i586-linux
export SUIFHOME=/usr/local/suif
export SUIFPATH=$SUIFHOME/$MACHINE/bin:/usr/bin:/usr/local/bin
export COMPILER_NAME=gcc
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:$SUIFHOME/$MACHINE/solib
export PATH=${PATH}:$SUIFHOME/$MACHINE/bin
```

Figure A.16: Environment variables that need to be set before using VSSC

The **LD_LIBRARY_PATH** environment variable is needed if any of the SUIF, Simple-SUIF, or VSSC systems are compiled as shared libraries on your system.

A.4.5 Using VSSC

The VSSC system allow an infinite numbers of optimizations to be used with it. For each optimization, the optimization writer registers a procedure with the following *signature* with the VSSC system:

```
simple_instr *procedure_name(simple_instr *inlist, char *procedure_name)
```

Registering your procedure is done by calling (defined in **vssc_simple.h** and described in Sections A.4.2 and A.4.6):

```
void vssc_register_opt(char *name, char *flagname,
             simple_instr *(*procedure)(simple_instr *, char *));
```

The code in Figure A.17 (derived from **main.cc** in the example skeleton files mentioned in section A.4.6) demonstrates the registration of an optimization routine for dead code elimination:

```
#include <stdio.h>
#include <vssc_simple.h>

simple_instr *dead_code(simple_instr *inlist, char *procedure_name)
{
   GUI->set_procedure_name(procedure_name);
   GUI->set_basicblock_number(0);
   GUI->set_graph_type("Flow Graph");

   FlowGraph *FG = new FlowGraph(inlist, procedure_name);

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   return inlist;
}

int main(int argc, char *argv[])
{
   vssc_register_opt("Dead Code Elimination", "deadcode", dead_code);
   vssc_init_suif(argc, argv);
}
```

Figure A.17: Sample optimization registration

As you can see from the example code above, the optimization writer's registered procedure is given by the VSSC system, a NULL terminated (last element's **next** pointer is NULL) linked list of Simple-SUIF instructions and the name of the procedure these instructions belong to. The user can them modify the linked list in any valid manner (such as performing useful optimizations!). The procedure must return a NULL terminated linked list of Simple-SUIF instructions back to the VSSC system. This linked list is of the same format and organization as the one passed in. If a non-NULL terminated or malformed linked list of Simple-SUIF instructions is returned, the VSSC system will get confused and most likely quit with an error.

A.4.6 Your First VSSC Optimization

To start writing your own optimization routines, download via anonymous ftp, `ftp://ftp.cs.ucr.edu/pub/publications/thesis/brian_harvey/skeleton.tar.gz`, which creates a new subdirectory for you which contains a `Makefile`, a `main.cc`, and several other files to start you off. The `Makefile` makes use of the SUIF makefiles to make it less complicated. The sample `Makefile` is shown in Figure A.18.

```
#
# Sample Makefile to generate your own VSSC compiler
#
#
# change 'mycompiler' to whatever you want.
#
TARGET =        mycompiler
PACKAGE_NAME =  mycompiler
SUPER_PACKAGES_PATH = vssc

# Don't change this unless you know what you're doing!
LIBS =          -L/usr/X11/lib -lvssc -lsimple -lsuif -lX11 \
                -ltk4.1 -ltcl7.5 -ltcldot -ldl

# List all .c files here
CSRCS =

# List all .cc files here
SRCS =          BasicBlock.cc FlowGraph.cc main.cc

# List all .o files here. (Each .c and .cc file has a .o file)
OBJS =          BasicBlock.o FlowGraph.o main.o

EXTRA_CFLAGS =  $(INCLDIRFLAG)

all:            prog

include $(SUIFHOME)/Makefile.std
```

Figure A.18: Sample Makefile

Typing **make** will compile your code (those files listed in the **SRCS** and **CSRCS** sections in the Makefile) and generate your very own VSSC compiler (called **mycompiler** in this example **Makefile**).

In `main.cc` (similar to the example code shown in the previous section), a sample optimization procedures are declared and registered with the VSSC system. All you need to do is write the optimization routines! Figure A.19 shows the contents of `main.cc`:

```
#include <stdio.h>
#include <vssc_simple.h>

#include "BasicBlock.h"
#include "FlowGraph.h"

simple_instr *dead_code(simple_instr *inlist, char *procedure_name)
{
   FlowGraph *FG;

   /* Set what the procedure name is and what basicblock we're looking at. */

   GUI->set_procedure_name(procedure_name);
   GUI->set_basicblock_number(BLANK);
   GUI->set_graph_type("Flow Graph");
   fprintf(stderr, "Doing deadcode elimination\n");

   FG = new FlowGraph(inlist, procedure_name);

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   /* Optimization code */

   GUI->step();

   /* Etc. */

   return FG->instructions_head;
}

simple_instr *cse(simple_instr *inlist, char *procedure_name)
{
   fprintf(stderr, "Doing common subexpression elmination\n");

   return inlist;
}


int main(int argc, char *argv[])
{
   vssc_register_opt("Local Common Subexpression Elimination", "cse", cse);
   vssc_register_opt("Dead Code Elimination", "deadcode", dead_code);

   vssc_init_suif(argc, argv);
}
```

Figure A.19: Sample `main.cc`

NOTE: The skeleton code provided is meant to be an example only. It is very incomplete and will not run correctly as is. It is provided to you as an example to help to get started and to give you some idea how you might want to organize your code.

After you've taken the skeleton files and written some code, try compiling it, and run your newly created VSSC compiler (called **mycompiler** in the example `Makefile` but you can change the name if you want to) with a C file:

```
mycompiler -v test1.c
```

The **-v** option indicates that you want to bring up the VSSC GUI. Your VSSC compiler can also run on a non-graphical mode. See Section A.5.3 for more details.

When you are done using your VSSC compiler, a file will be left in the current directory with the name `<file>.suif`. This binary file is in the SUIF intermediate format and its contents can be printed out by the **printsimple** command. See Section A.5.2 for more details.

A.5 VSSC Tips

The following tips are provided to make writing your compiler optimizations using VSSC easier.

A.5.1 Debugging VSSC Optimizations

At some point while you're writing VSSC optimizations, you may find that you need to run your optimizations through a debugger like GDB. Because of the way VSSC is implemented, gdb won't be able to step through your optimization. If your optimization happens to cause a segmentation violation (segfault), VSSC will catch it and gracefully quit notifying you of the problem. To use gdb with your VSSC compiler, run your VSSC compiler with the `-d` option. This lets the VSSC system know that you want to debug your code and gdb will be able to step through your optimization. The only limitation when using the `-d` flag is that the VSSC system can only perform one optimization pass. If you try to perform a second optimization, an error will be generated and your VSSC compiler will quit.

A.5.2 printsimple

printsimple is a program provided by the Simple-SUIF distribution. It can be used to print out the contents of the Simple-SUIF files produced by VSSC (those files ending in **.suif**). An example of output is shown in Section A.3.3.

A.5.3 Non-GUI VSSC

A VSSC compiler has the capability to be run in a non-graphical mode. Invoking the compiler without the **-v** flag causes it to not run in graphical mode. However, in order to have the compiler perform optimizations, these optimization must be specified on the command-line. The **-O** *optname* flag is used to specify an optimization to be run. *optname* is the same string that the optimization writer used for the second parameter of **vssc_register_opt**. An infinite number of **-O** flags can appear during the invocation of the VSSC compiler in non-graphical mode. For example, the following command performs deadcode elimination, followed by common sub-expression elimination, followed by deadcode elimination again (`deadcode` and `cse` are the command-line flags specified when the deadcode elimination and common sub-expression elimination optimizations were registered with the VSSC system by the optimization writer):

```
mycompiler -O deadcode -O cse -O deadcode quicksort.c
```

As was mentioned previously, the VSSC compiler produces a SUIF format file which contains the results of all the optimizations that were performed.

A.5.4 Making Assertions

It is good programming practice to use **assert** statements in your code. Assertions, which act as an error checking mechanism, are tests for things that never should occur the program was executing correctly. If an assertion fails during program execution, the program immediately aborts. SUIF provides two extended versions of the C **assert** statement which are shown in Figure A.20.

assert(*expression***)** SUIF overrides the **assert** statement provided in the Standard C library. This version prints out more detailed information about where the assertion failed.

assert_msg(*expression*, **(***printf message***))** Besides an expression to be tested, like **assert**, **assert_msg** also allows a printf style message for the second parameter (don't forget the parenthesis around the second argument!). This message is printed out if the assertion fails. For example:

```
assert_msg(curr_instr != NULL,
           ("curr_instr is NULL in procedure %s", procedure_name));
```

Figure A.20: Assertions provided by SUIF

A.5.5 Using Data Structures

If you didn't already know, writing compiler optimizations (or any large project) requires that you manage a lot of data in data structures. Commonly used data structures in compiler optimizations include lists, arrays, bitsets, and graphs. While it's possible for you to write your own data structures and routines to manage them, this task can be time-wasting and tedious. You would have to design, implement, and debug everything yourself. Usually, it is easier to use a data structure library that has been developed and tested by someone else.

Fortunately, the SUIF libraries include most of the commonly used data structures needed in compiler optimizations so you don't have to write them yourself. These data structures and examples of their use are shown below. Most the data structure classes provided by SUIF are meant to be base classes. They have no information about what the type of the data they are supposed to contain. In most cases, you have to create a derived class for a particular data item you with that data structure to contain. For example, the first class described below, `glist`, is useless if used as is. However, you can create a derived class which implements a generic linked-list list of whatever data item type you wish (eg list of instruction structures or list of basic blocks). SUIF provides powerful macros which create these derived classes for you. Simply specify the *type* of the data item you want that data structure to store and a macro creates a new class for you automatically. These macros, which are provided for most of the SUIF data structure classes, perform the same functionality as C++ templates. In each data structure description below, we show an example using the macro for the data structure (if one exists) to create a new class and then exercise some of the commands for that class by adding, searching for, and deleting a data item in a instance of the new class that was created.

The macros also generate a method for iterating through the items in the data structure class it generates. Using the iterator, you can easily step through the data structure if you need to perform the same action on each element in the data structure. Examples of how to use this iterator is also shown in some of the examples.

These data structures, shown in Figures A.21, A.22, A.23, A.24, and A.25, are described in more detail in Chapter 11, Generic Data Structures, of the document "The SUIF Library Version 1.0" [2]. You can also find out more about each data structure class (including API calls not listed below because they are not commonly used) by looking at the actual header file for the class in `$SUIFHOME/include/suif/`*class*`.h`. To use these data structures and their macros in your code, you just need to `#include <suif.h>`.

SUIF also provides other data structure classes such as hash tables and move-to-front lists. Information about them can be found in their header files and the SUIF library documentation mentioned above.

[2] http://suif.stanford.edu/suif/docs/suif_91.html#SEC91

Data Structure: Generic List

Class Name: `glist`

Description: Implemented as a singly linked list, this is a generic list class.

Macro: `DECLARE_LIST_CLASS`

API:

```
boolean is_empty() const
glist_e *head() const
glist_e *tail() const
glist_e *push(glist_e *e)
glist_e *pop()
glist_e *append(glist_e *e)
glist_e *insert_before(glist_e *e, glist_e *pos)
glist_e *insert_after(glist_e *e, glist_e *pos)
glist_e *remove(glist_e *e)
void erase()  /* deletes items from list */
int count()
boolean contains(const glist_e *e)
glist_e *operator[](int ndx)
```

Example:

```
DECLARE_LIST_CLASS(NodeList, GraphNode *);
NodeList nodes;

/* Add item */

GraphNode *tmp;
nodes.append(tmp);

/* Search for item */

if (nodes.contains(tmp))
{
   /* Found it! */
}

/* Remove item */

nodes.remove(tmp);
```

Figure A.21: Generic List class

Data Structure: Association List

Class Name: `alist`

Description: An element in an association list contains both a key and data pointer. The data associated with a key can be retrieved with a simple lookup method. There exists no macro for this data structure but one really isn't need. The data and key fields in an association list item are of type `void *` so the *key* can be of any type as well as the type of what the *data* represents.

Macro: No macro

API:

```
alist_e *head()
alist_e *tail()
alist_e *push(alist_e *e)
alist_e *pop()
alist_e *remove(alist_e *e)
alist_e *enter(void *k, void *i)
alist_e *search(void *k)
void *lookup(void *k)
boolean exists(void *k, void **i = NULL)
```

Example:

```
alist nodes;

/* Add item */

tmpnode = new GraphNode;
tmpnode->title = strdup(name);
nodes.enter(tmpnode, NULL);


/* Search for item */

alist_iter node_iter(&nodes);
alist_e *item;
while (!node_iter.is_empty())
{
  item = node_iter.step();
  if (!strcmp(((GraphNode *)item->key)->title, name))
     /* Found it! */
}

/* Remove item */
alist_e *item = new alist_e(node, node);
nodes.remove(item);
```

Figure A.22: Associative List class

Data Structure: Doubly-Linked List

Class Name: `dlist`

Description: Essentialy the same as the generic list except it is implmented as a doubly-linked list.

Macro: `DECLARE_DLIST_CLASS`

API: Same API as generic list

Figure A.23: Double Linked List class

Data Structure: Bit Vector

Class Name: `bitset`

Description: Standard bitset class (bit vector representation is implemented as a doubly-linked list for a set of integers).

Macro: no macro

API:

```
bit_set()  { first = 0; last = 0; bits = NULL; }
bit_set(int f, int l, boolean no_clear = FALSE);
~bit_set() { delete bits; }

void expand(int f, int l, boolean no_clear = FALSE);
int lb()   { return first; }
int ub()   { return last; }
void clear();       /* clear all the bits to 0 */
void universal();   /* set all bits to 1 */
void add(int e);    /* set bit e to 1 */
void remove(int e); /* reset bit e to 0 */
void invert();      /* invert all bits */
boolean contains(int e);
void set_union(bit_set *l, bit_set *r);
void set_intersect(bit_set *l, bit_set *r);
void copy(bit_set *s);
void transfer(bit_set *src, boolean del = TRUE);
void operator=(bit_set &b)      { copy(&b); }
void operator+=(bit_set &r);    /* bit-wise OR */
void operator*=(bit_set &r);    /* bit-wise AND */
void operator-=(bit_set &r);    /* bit-wise subtraction */
boolean operator==(bit_set &r); /* bit-wise comparison */
boolean operator!=(bit_set &b)  { return !(*this == b); }
boolean operator<=(bit_set &r); /* subset of the bits? */
boolean operator^(bit_set &r);  /* test for non-empty is implmented as a doubly-linked list
                                   intersection */
boolean is_empty();             /* all zeros? */
boolean is_universal();         /* all ones? */
int count();                    /* count the 1 bits */
void print(FILE *fp = stdout, char *fmt = "%d,");
```

Figure A.24: Bit set class

Data Structure: Extensible Arrays

Class Name: x_array

Description: An array class that has no size limit. This class implements the [] operator so that elements of the array can be accessed just like in regular C arrays (ie A[5]).

Macro: DECLARE_X_ARRAY

API:

```
x_array(int sz); /* sz is initial size. */
~x_array();

void *& operator[](int i);
int extend(void *e);
int ub(); /* Return number of elements in array. */
```

Example:

```
DECLARE_X_ARRAY(NodeArray, GraphNode *);
NodeArray nodes;

/* Add item */

GraphNode *tmp = GRAPH->addNode("B2", NULL);
nodes.extend(tmp);

/* Search for item (GraphNode *tmp2) */

for (int i=0; i<nodes.ub(); i++)
{
   tmp = (GraphNode *)nodes[i];
   if (tmp == tmp2)
   {
      /* Found it! */
   }
}

/* Remove item: You can't really remove an array index but you can
   remove what it points to (after you find it first) leaving
   a "hole" in the array. */

/* Remove GraphNode *tmp3 */

for (int i=0; i<nodes.ub(); i++)
{
   tmp = (GraphNode *)nodes[i];
   if (tmp == tmp3)
   {
      /* Found it. */

      delete(tmp);
      nodes[i] = NULL;
   }
}
```

Figure A.25: Extendible array class

A.5.6 Examples using SUIF data structures and VSSC API

This section contains several examples using the SUIF data structures and the VSSC API commands. The code given in these examples is not meant to work by itself and some of it may be pseudo-code. It is provided only as an example to help you get comfortable writing your own optimizations in VSSC.

Example 1

In this example, shown in Figure A.26, we define a simple symbol table class that uses the generic list data structure as a data structure for all the symbols. The incomplete class shown exercises some of the operations you can perform on the generic list. In your optimizations, you probably won't need a symbol table class. This is just an example!

Example 2

In this example, shown in Figure A.27, we create an extensible array of pointers to basic blocks. For each basic block we find we add it into the VSSC text widget as well as inserting a node representing it into the VSSC graph widget. Then we go through the array to print out the contents of each basic block.

```
class SymbolTable
{
   private:
      DECLARE_LIST_CLASS(SymTab, SymbolTable_Entry *);
      SymTab T;
   public:
      void enter(SymbolTable_Entry *e);
      void remove(SymbolTable_Entry *e);
      void print();
};

void SymbolTable::enter(SymbolTable_Entry *e)
{
   if (T.contains(e))
      printf("Error: symbol table already contains item");
   else
      T.append(e);
}

void SymbolTable::remove(SymbolTable_Entry *e)
{
   if (!T.contains(e))
      printf("Error: symbol table already contains item");
   else
      T.remove(e);
}

void SymbolTable::print()
{
   SymTab_iter node_iter(&T);
   SymTab_e *item;
   while (!node_iter.is_empty())
   {
      item = node_iter.step();
      printf("Symbol: [%s]\n", item->name);
   }
}
```

Figure A.26: Example 1 source code

```
#include <vssc_simple.h>

class BasicBlock
{
   public:
      int id;      /* BasicBlock number  (eg BasicBlock 4)
        .
        .
        .
};

DECLARE_X_ARRAY(BasicBlocks, BasicBlock *);

BasicBlocks BBs;  /* BBs is an extensible array of pointers to
                     BasicBlock items */

int main(int argc, char **argv)
{
   foreach Basic Block BB that we find
   {
      BBs.extend(BB);

      /* Add basic block to end of text widget with no animation */

      GUI->add_basicblock(BB, NULL, FALSE);

      /* Add node to graph representing basicblock and add a
         dataitem for that node which contains the id of the
         BasicBlock

         "BasicBlock: 4" */

      char node[10];
      sprintf(node, "B%i", BB->id);
      GraphNode *BBnode = GRAPH->addNode(node, FALSE);
      sprintf(node, "%i", id);
      GRAPH->addDataItem(BBnode, "BasicBlock:", node);

   }

   /* foreach Basic Block */

   for (i=0;i<BBs.ub(); i++)
   {
      BasicBlock *BB = (BasicBlock *)BBs[i];

      /* Print out instructions in basicblock. */

      simple_instr *instr;
      instr = BB->head();
      while (instr != BB->tail)
      {
          simple_instr_print(stdout, instr);
          instr = instr->next;
      }
      simple_instr_print(stdout, instr);  /* Tail instruction */
   }
}
```

Figure A.27: Example 2 source code