Machine-Independent Compiler Optimizations for the U of T DSP Architecture

by

Sanjay M. Pujare

A thesis submitted to fulfil the equivalent of three courses of the requirements for

the Degree of Master of Engineering in the

Department of Electrical and Computer Engineering

University of T oronto

CANADA

Sanjay M. Pujare

University of Toronto

Department of Electrical Engineering

Machine-Independent Compiler Optimizations for the U of T DSP Architecture

Abstract

There are two kinds of compiler optimizations: machine-dependent optimizations and machine-independent optimizations. For specialized architectures such as a DSP architecture, machine-dependent optimizations play a very important role, because these optimizations seek to exploit the special hardware features of the target processor to maximize performance. However, using these machine-dependent optimizations alone in a compiler requires that programmers write their programs in a specific style. It is proposed that certain machine-independent optimizations will allow programmers to realize the benefits of machine-dependent optimizations without imposing overly restrictive rules on the programming style.

An existing SUIF-based C compiler is retargeted to the U of T DSP architecture and extended to include several machine-independent optimizations such as induction variable creation, loop-invariant code motion and dependence analysis. Two versions of several existing DSP benchmarks are written in a natural C-style. The augmented compiler is used to compile these versions and the performance of the generated code is analyzed and compared with the performance of the code generated from the original version that was written in the specific style dictated by the earlier GCC-based compiler. For seven out of twelve kernel benchmarks and for five out of six application benchmarks, the performance of one of the new versions of the benchmarks is equal to or better than the performance of the original version, when all versions of the benchmarks are compiled by the augmented compiler.

Acknowledgements

I would like to express my gratitude to my supervisor Professor Corinna Lee for the direction, guidance, and encouragement she provided during the course of this project. It is only because of her ideas and her help that I have been able to do most of the things I did in this project. I would also like to thank Professor Todd Mowry for giving me a good foundation on compiler optimizations and for all the SUIF-related information and advice he gave me.

My thanks go to Mark Stoodley for all the post-optimizer related help I got from him, to Mazen Saghir for all the DSP simulator related and general help, to Alan V i Cuong Chan for the help with FrameMaker and other things, and to others in LP392 whom I came to know during my project.

Finally, I could not have completed this project without the support and encouragement I got from my family and friends through some difficult times; my heartfelt thanks go to them.

Machine-Independent Compiler Optimizations for the U of T DSP Architecture

Contents

List of Figures

Chapter 1 Introduction

DSPs (Digital Signal Processors) are specialized processors whose architectures are tuned towards running high performance numerically intensive applications in real time. Unfortunately DSP application developers have tended to program these processors in their assembly languages because of the poor performance of the code generated by traditional compilers for high-level languages (HLLs) such as Fortran or C. The objective of this project is to explore different optimizing algorithms and to implement a compiler and its optimizing passes that will generate high quality and efficient code for a target DSP architecture and thereby demonstrate the effectiveness of optimizing compilers in DSP environments.

This chapter describes the motivation behind this project work, outlines some objectives and summarizes the methodology of this work. The organization of this write-up and contributions of this project are also presented.

1.1 Motivation and Objectives

As DSP applications grow in sophistication and complexity, DSP programmers would like to use good optimizing HLL compilers rather than the assembly language of their DSP processor. However the poor performance of compiler generated code has prevented widespread use of HLLs by DSP programmers.

Clearly there is a need for optimizing HLL compilers that can be used by DSP programmers. Some of the reasons for this are: HLL compilers improve the usability of high-performance DSP processors by allowing application developers to program these computers in a high-level language, and thus to concentrate on their problem domain rather than architectural or machine programming issues; HLL compilers complement the hardware in achieving high-performance because they usually use global information of the program to generate code that can effectively use architectural features such as registers and multiple functional units; developing DSP algorithms in a HLL has obvious benefits such as portability, readability and maintainability[TjH92].

Based on this need for optimizing HLL compilers, earlier research carried out before this project concentrated on the DSP architecture and a retargetable compiler with machine-specific optimizations[Sin92]. As a result of these efforts, a model DSP architecture and corresponding compiler were developed that allowed writing of DSP programs in a high-level language, namely C. The resulting compiler system consists of a retargeted GCC compiler and a post-optimizer that performs the machine-specific optimizations[Sin92][Sag93]. Unfortunately the compiler imposes some constraints on the way the program must be written to achieve the desired performance gains of the DSP architecture[Sag93]. The constraints that are imposed by this compiler and that forced a specific programming style on the source code are clearly undesirable. Moreover, these programs could not really be called ‘higher level’ in nature with restrictions such as the use of pointers to access arrays.

This project has been largely motivated by a desire to improve upon the existing compiler system and to explore new optimizations to eliminate the restrictions imposed by our earlier compiler system. Given these motivations, some of the objectives of this work are:

- retargeting of a suitable compiler system (SUIF) to our DSP architecture that would allow research into some of the optimization areas that we think are important for the DSP architectures;
- enhancement of this compiler system for integration into the existing development tools for the DSP architecture;
- exploring new optimization algorithms;
- modification of benchmarks to reflect the true 'higher level' nature of the HLL programs; and
- comparison and analysis of the performance of code generated by the new compiler system with the original benchmarks and the modified benchmarks.

1.2 Contributions

Some primary contributions of this project are as follows:

- Development of a SUIF code generator (DGEN) for the DSP model architecture.
- Addition of new optimizations to the SUIF compiler system, so that 'true' HLL programs can be compiled to generate code with performance that matches the performance of 'DSP-style' programs. Namely the following passes were developed to enhance the performance: induction variable creation (INDV AR), array common sub-expression elimination (part of INDV AR), and dependence analysis (DEP_AN).
- Development of a new ***lex-*** and ***yacc*** -based front-end for the post optimizer that is more flexible and that reads SUIF annotations.
- Creation of two versions of existing DSP benchmarks to utilize the optimizations being performed by the compiler .
- Analysis of the performance of all the benchmarks and comparison of SUIF-generated code versus GCC-generated code.
- Suggestions for future work based on conclusions derived from this work.

1.3 Organization of the Report

The remainder of this report contains six chapters. Since this project was a continuation and extension of the previous work done on the DSP architecture and its compiler , Chapter 2 presents some background information on these. It also explains the idea of a 'high-level language' in the context of our DSP architecture and the benchmarks and discusses how the issue of 'high-level language' has motivated this work. The retargeting of the SUIF compiler system to the DSP architecture first required modifying the code generator of the SUIF compiler . Chapter 3 describes the code generator work done for the SUIF compiler . Chapter 4 describes the implementation of various optimization passes that were written specifically to achieve the desired performance goals of the compiler . Some background information on the existing SUIF optimization passes and results of the discussion with some people in the SUIF developer/ user community are also presented to explain the reasons that led to the development of these optimization passes. Chapter 5 describes the modifications made to the post-optimizer front-end to make it more flexible and to enable it to read SUIF annotations. The performance of the code generated by the SUIF-based compiler vis-a-vis that by the GCC-based compiler is presented in Chapter 6. It also discusses the impact of the optimization passes and discusses problems, features or drawbacks of the code generated by the SUIF compiler . Finally Chapter 7 summarizes the work of this project, and presents its conclusions. It also discusses some areas of the compiler in which more work can be done to still improve the performance of the code generated.

Chapter 2 Background

This chapter presents some background material so that the reader is better able to understand the issues involved and the context in which the project work was done. This project work was a continuation of the previous work done in this area and used that as a foundation for this new work. Specifically this work used the post-optimizer *OptimizeDSP*, the dsp simulator ***dspsim***, and the GCC-based compiler ***gcc-DSP*** – tools that were developed in the previous projects. The interested reader is requested to refer to this work [Sin92] [Sag93] in addition to this background.

2.1 Model DSP architecture and the DSP simulator

This work was part of a larger ongoing research project which seeks to develop a DSP architecture and a related compiler together, so that the DSP architecture would serve as a better target for HLL compilers and the compiler would actually demonstrate the effectiveness of this architecture by generating efficient code for it. Figure 2.1 shows the overall set-up of this DSP development environment and identifies the changes we made to the system as part of this work.

As stated in Chapter 1, DSP processors are especially tuned or designed to enable fast and efficient execution of DSP applications. A variety of requirements that characterize a DSP architecture are as follows [Sin92]: MAC instructions, low overhead looping, increased memory bandwidth, addressing support for modulo and bit-reversed addressing, hardwired control circuitry (as opposed to microprogrammed), and multiple functional units to exploit DSP parallelism. Very Long Instruction Word (VLIW) is a form of architecture that is used for supporting multiple functional units and is found to be quite suitable for a DSP processor [Cas94]. Figure 2.2 shows the U of T DSP architecture with its nine functional units.

However it is not enough to have only a specialized architecture for DSP applications. Optimizing compilers have become an essential part of modern high-performance computer systems. For any machine architecture feature to be useful, it has become necessary to use an optimizing compiler that can exploit this feature in the code that it generates. As optimizing compilers become more effective, programmers can become less concerned about the details of the underlying processor architecture and can employ higher-level, more succinct and more intuitive programming constructs and program optimizations. At the same time, hardware engineers can aim for greatly improved performance from their machines with fewer constraints on their designs, since they need to only concern themselves with the suitability of their architecture as a compiler target rather than as a direct programmer interface.

The research and project work [Sin92][Sag93] that was the precursor of this project explored these issues about DSP processors and HLL compilers for these processors. It designed a general-purpose DSP architecture along with, and as a target of, an HLL compiler, so that the compiler could effectively utilize the DSP-specific features of the target architecture. See Section 2.2 for more details.

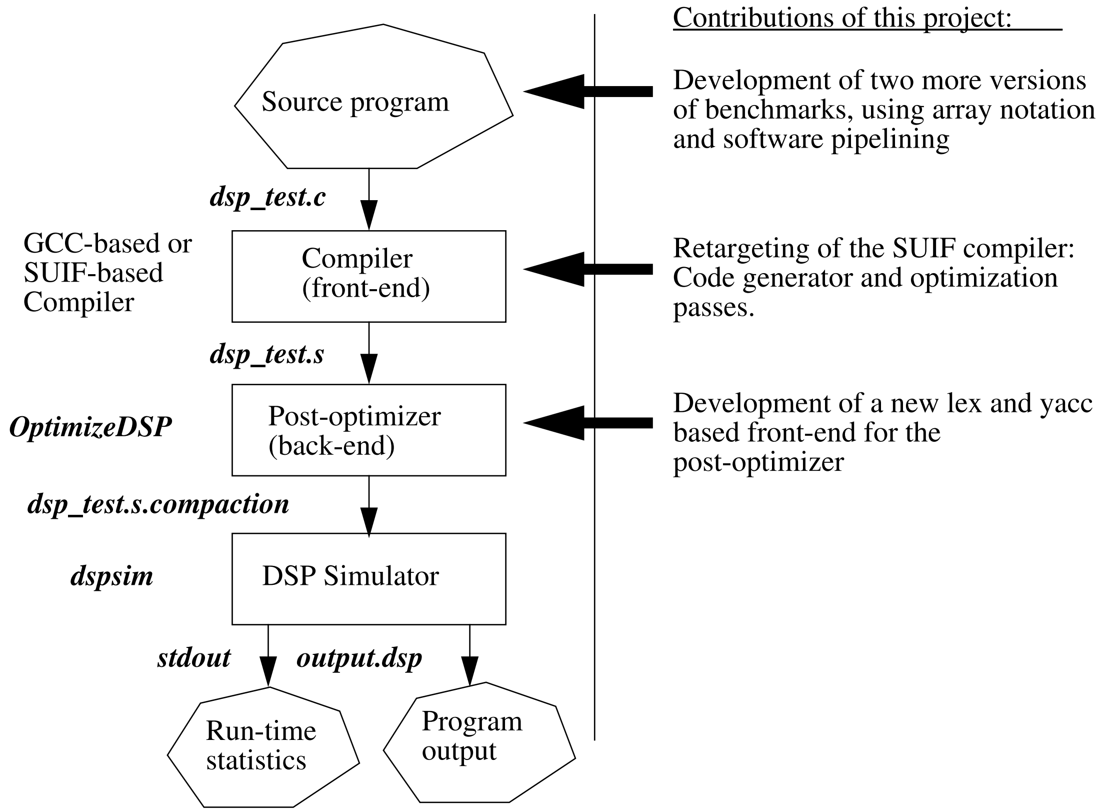

Figure 2.1 DSP tools set-up and contributions of this project

The designing of the HLL compiler and the target machine architecture together resulted in efficient code generation, and demonstrated that this system performed almost three times better than the best commercial system of that time [Sin92]. The architecture itself was implemented not as a piece of hardware but as an instruction set simulator, a program that would read the assembly language output of the compiler and simulate its execution on the DSP processor. This program could also generate statistics on the program execution which could be used to analyze code performance. The initial version of the architecture did not support the floating point type. However this was added later by Mazen Saghir while working on his Master's thesis [Sag93].

2.2 Optimizing DSP C Compiler and Post-optimizer

The original implementation, done by Vijaya Singh as part of his Master's thesis work [Sin92], was a port of the GNU-project C compiler (GCC) to the model DSP architecture. The port was accomplished by modifying GCC's back-end to emit DSP processor code. It was precisely the portability of its back-end that made GCC an attractive compiler to port to the DSP architecture.

However, instead of incorporating all DSP-specific optimizations in the GCC compiler, it was decided to develop a 'post-optimizer' that would do these optimizations. So compilation would be a two-step process, whereby GCC would be used to emit assembly language code for the target machine as if it was an ordinary sequential instruction machine, and the post-optimizer would then take this code and compact it or perform instruction scheduling for the VLIW nature of the DSP architecture. In addition to this, the post-optimizer would perform

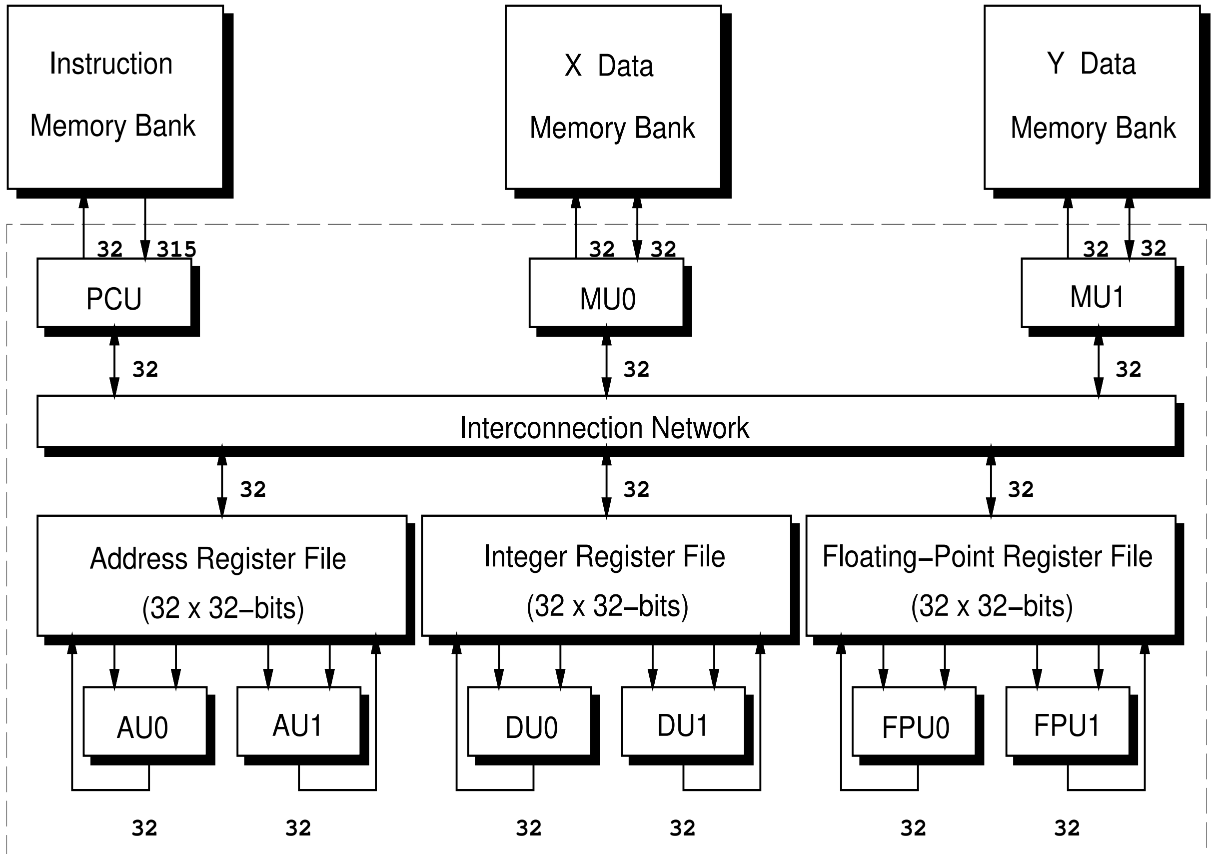

Figure 2.2 U of T DSP Architecture and its nine functional units

other DSP-specific optimizations such as hardware do-loop generation, modulo addressing (if necessary), and memory partitioning.

This original port of the GCC compiler by V ijaya Singh did not have floating point support, as the DSP architecture itself did not support floating point data type. However this was added later by Mazen Saghir to the compiler [Sag93]. He also modified the post-optimizer to include an enhanced version of the memory partitioning algorithm.

2.3 Limitations of the GCC-based compiler

Although the modified GCC-compiler demonstrated the benefits of a compiler that tries to utilize all features of our DSP architecture, it failed to apply certain optimizing transformations. This resulted in some constraints being imposed on the application programmers for them to realize the performance gains offered by the DSP architecture and the compiler . These constraints came to be called the ‘DSP-style’ of writing code. Efficient code-generation by the compiler plus the post-optimizer system required that source programs be written in the ‘DSP-style’. Some characteristics of the DSP-style are:

- Use of pointers to access arrays, especially inside loops. Special pointer variables should be used to access arrays inside loops instead of indexing them with integer expressions. For example, the following fragment of C code:

```
int a[N], b[N];
for (i = 0; i < N; i++)
{
```

```
        b[i] = a[i];
}
```

which looks like natural and readable C code has to be modified as shown below to conform to the DSP-style:

```
int a[N], b[N], *pa, *pb;
pa = a; pb = b;
for (i = 0; i < N; i++)
{
      *pb++ = *pa++;
}
```

This requires the use of additional variables and is less readable.

- Use of software-pipelining to increase the amount of parallelism exploited by the post-optimizer . The above example would be re-written as:

```
int a[N], b[N], *pa, *pb, tmp;
pa = a;
pb = b;
tmp = *pa++;
for (i = 1; i < N; i++)
{
    *pb++ = tmp; tmp = *pa++;
}
*pb++ = tmp;
```

The instruction scheduler (of the post-optimizer) can generate a one-instruction loop in this case as opposed to the two instruction loop in the previous example. This is because, the usage of variable **tmp** achieves a limited form of software-pipelining – **tmp** allows the memory load of the next iteration and memory store of the current iteration to be done concurrently (note that if a register is both read and written in an instruction cycle, the read operation will read the old register value, before the register store occurs in that instruction cycle). T o describe this in technical terms, two operations which have a true dependency between them cannot be executed concurrently , whereas if the dependency was an anti-dependency then those two operations could be executed concurrently .

- Usage of temporary variables by the application programmer to eliminate re-access of array elements, as shown in the following example (unrelated to the above examples):

```
int a[N], *pa, tmp0;
pa = a;
for (i = 0; i < N; i++)
{
    tmp0 = *pa++;
    tmp2 = tmp0 + 3;
    tmp3 = tmp0 * 4;
}
```

Instead of accessing **`*pa`** twice, a temporary variable **`tmp0`** is used to store the value of **`*pa`**.

In addition to the idiosyncracies of the DSP-style, the GCC-based compiler lacked certain features that would be useful in some of the new optimizations that are being explored in the DSP architecture. For example, the GCC-based compiler does not have the ability to annotate the code it generates. These annotations typically could be used by the front-end compiler to pass on useful information such as the output of data flow analysis to either the post-optimizer or even the DSP simulator . Also the GCC compiler does not have a dependence analyzer that would give useful information about array accesses for optimizations such as software pipelining, which is a more sophisticated form of instruction scheduling.

2.4 An alternate compiler - SUIF

In view of the limitations of the GCC-based compiler , it was decided to retarget a new compiler for our model DSP architecture, so that a number of different optimizing algorithms could be implemented and their impact on the resulting code investigated. As stated in the previous chapter , the GCC-imposed limitations were the main motivation behind this project.

The SUIF (Stanford University Intermediate Format) compiler was chosen as the compiler that would be retargeted to our DSP architecture and the bulk of work that resulted from this retargeting effort is what constitutes this project. The SUIF compiler system is a flexible framework for advanced compiler research, specifically in the area of optimizations for high performance platforms such as our DSP architecture. The SUIF compiler has become available only recently and hence could not be considered earlier for retargeting when Vijaya Singh [Sin92] ported the GCC compiler .

SUIF offers various advantages that make it an ideal compiler to work with for a project such as this. SUIF defines an intermediate format for the compiler that is capable of a higher level representation, such as abstract syntax trees, as well as a lower level representation such as flat instruction lists or a mixture of both. The SUIF library provides an object-oriented implementation of the SUIF format. This has been written in C++ [Str92], and it defines classes to represent all the various elements of the intermediate format and to perform some common operations on them [SUL94].

The SUIF compiler is implemented as a set of compiler passes built on top of the kernel that defines the SUIF intermediate format implemented in the SUIF library . Each pass typically performs a single analysis (or in some cases multiple analyses) and then writes the results to a file. Since all the passes use SUIF format as their input as well as output formats, these passes can be re-ordered or new passes can be inserted at almost any point in the compilation. This is what makes SUIF a flexible compiler and allows its users to explore new optimization techniques in various combinations.

The other features of the SUIF compiler system that we found attractive and that we did not find in the GCC compiler were the availability of the dependence analysis support, the ability to attach annotations to the output code, and the fact that there has been on-going research in the area of compilers for parallel and high-performance machines, where the SUIF compiler is used.

In the next chapter we will look at DGEN, the code generator for our DSP processor; which is the first step in retargeting SUIF to the DSP architecture.

Chapter 3 DGEN - Code Generator

Retargeting a new compiler such as SUIF to the model DSP architecture would require, as the first step, writing a new code generator or modifying an existing one that will generate code for the new target machine. The original SUIF compiler came with only one code generator: MGEN which generated code for the MIPS R3000 chip. We decided to modify this into DGEN to make it generate code for our DSP architecture.

Figure 3.1 is a schematic representation of the structure of the SUIF compiler and all its phases. It also identifies the location of DGEN in the whole compiler. The SUIF compiler of Figure 3.1 is an expansion of the box labelled "front-end compiler" in Figure 2.1 which depicted our entire DSP development/ simulator environment.

This chapter starts with a discussion of the issues involved in modifying MGEN to get DGEN. The discussion includes a brief overview and the structure of the SUIF compiler. This is followed by the description of actual changes that gave us DGEN from MGEN. Next we describe additions made to the DGEN code due to the DSP-specific nature of the target architecture. We conclude by describing a number of specific optimizations that we incorporated into DGEN to improve the performance of the generated code.

3.1 Porting Considerations

Before we describe how we updated the existing code generator for a new target, it is essential to understand the structure of this code generator and how it works. While reusing as much of the existing code as possible, we would like the updated code generator to utilize most of the special architectural features of the target DSP architecture.

This section presents a brief description of MGEN, the MIPS code generator, and identifies the parts of it that would need modification for retargeting.

3.1.1 Some SUIF background

This section presents some background information on the structure of SUIF and how MGEN fits into it, information that would be useful in understanding DGEN and the porting considerations [SUC94].

Since SUIF is a flexible compiler framework meant to help researchers in exploring new compiler optimization/ transformations techniques, the SUIF compiler is organized as a set of compiler passes which communicate with each other through the 'SUIF' intermediate code files – i.e. a pass generates its output in a file, which is used as input by the next pass. However because of historical reasons, there are two kinds of SUIF formats – old SUIF and new SUIF. The original SUIF system (now called "old SUIF") is obsolete, and the new SUIF system ("new SUIF" or just "SUIF"), which has been rewritten from scratch, has taken its place. Unfortunately some parts of the old SUIF system were not converted to new SUIF. These include the scalar optimizer ***oynk*** and the MIPS code generator called MGEN.

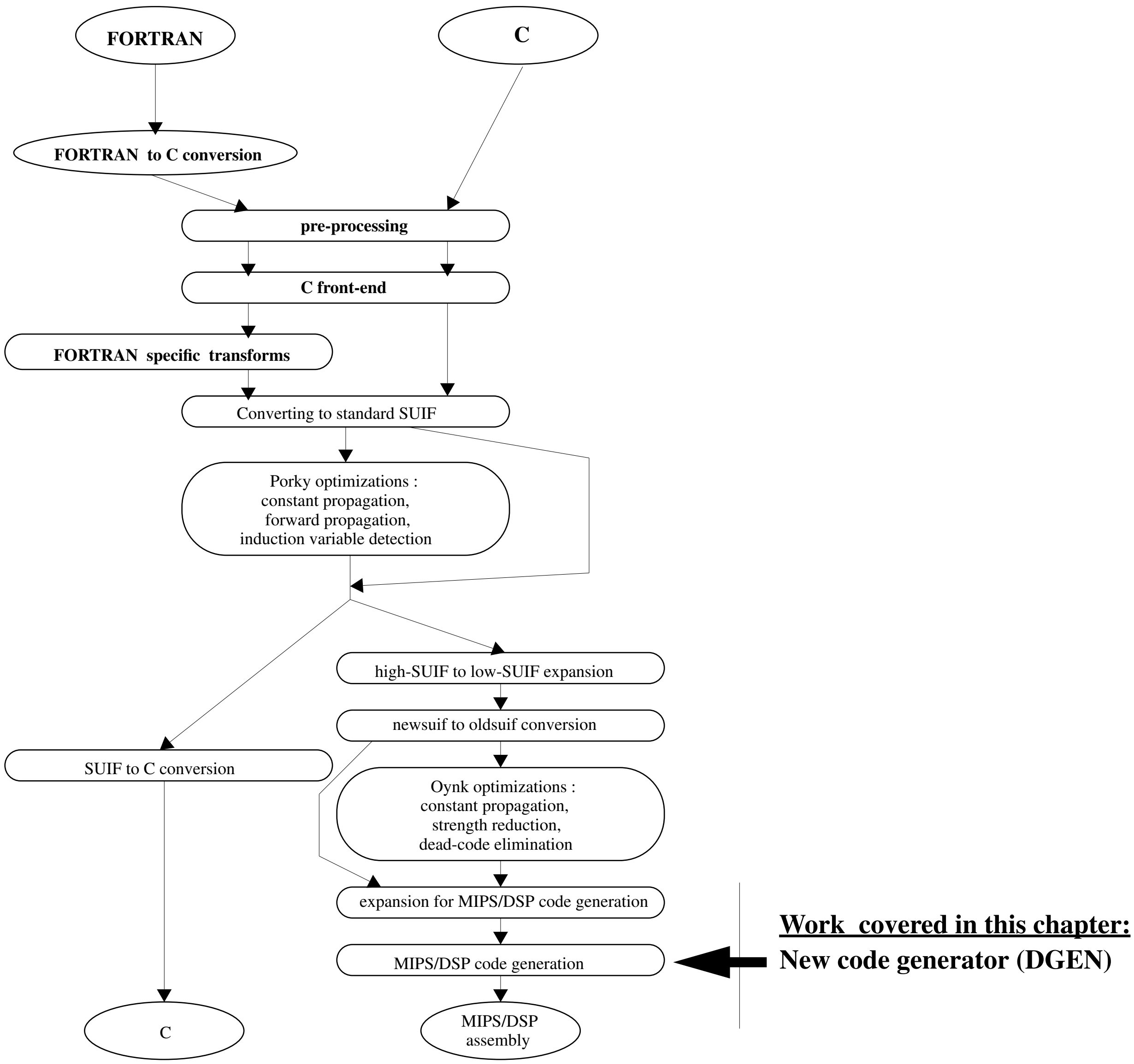

Figure 3.1 Structure and phases of the SUIF Compiler

Because the two file formats are different, SUIF comes with two conversion programs: ***oldsuif*** converts a new SUIF file to an old SUIF file, and ***newsuif*** converts an old SUIF file to a new SUIF file. However these conversion programs may not be successful in preserving all information in a file (e.g. annotations), because the representations of the two file formats are different.

The old SUIF format, which is what MGEN can read, can be briefly described as follows:

- Instructions are 3 operand instructions.
- Operands are one of pseudo-registers, node registers, hard registers or immediate values such as integers or labels. Pseudo-registers typically represent program variables which have not been aliased, and hence can be considered for optimizations. Node registers represent intermediate values of expressions, so each node register has exactly one definition and one use (with some very few

exceptions where MGEN has to expand SUIF instructions). Hard registers represent the actual registers of the target processor. Except for the code generator and the register allocator passes, most of the SUIF passes do not generate hard-register operands for the instructions.

- Instructions also have result types defined for them.

- There are other kinds of SUIF objects such as procedures, symbols, symbol tables and file-sets. A file-set consists of procedures while a procedure consists of symbol table(s) and instructions.

- Most SUIF objects can have annotations attached to them which convey additional information such as types or source level information.

On the other hand, the new SUIF format, which will be described in detail in Chapter 4, is based on Abstract Syntax Trees (ASTs), which typically represent control structures of the high level language, and expression trees. This representation is also called "high SUIF". Alternately all of the structures in the new SUIF format can be 'dismantled' to generate flat instruction lists, usually known as "low SUIF" format[SUL94].

Because of the existence of all these formats, even a simple compilation of a C file, with no special options, by the SUIF compiler can invoke many phases of the compiler. Many of these phases just convert from one format to another or perform dismantling of some high SUIF structures.

3.1.2 MGEN Structure

MGEN reads and translates one procedure at a time. MGEN has three passes that it runs on each procedure. Figure 3.2 shows a schematic block diagram of MGEN including its three passes. In this diagram, ovals represent input/output files, boxes represent functions or methods, and the three-sided open box represents a data-structure in memory.

Like the rest of SUIF, MGEN is written in C++ [Str92], and most of MGEN's code is implemented in the **`asmProc`** class. In addition it defines **`asmInstr`** class and **`ai_reg`** class to implement target instruction objects and target register objects respectively.

Following is the description of what each pass of MGEN does:

- Pass 1 translates SUIF instructions into assembly instructions. It also gathers information on all types of registers and the size of the stack frame.

- Pass 2 does register allocation or stack allocation (in case of spilling) of pseudo-registers, and stack allocation of spilled node registers. It also performs minor translation not done in pass 1 and certain peephole optimizations.

- Pass 3 does register allocation for node registers, and completes the translation of procedures by attaching header and trailer instructions to each procedure.

MGEN also has conventions with regard to the use of registers in the generated code. Thus it uses general registers **`d`**4 through **`d7`** and float registers **`f12`** through **`f15`** for passing parameters during a function call. It uses general register **`d2`** and float register **`f0`** for passing return values of function calls. It uses general register **`d29`** as the stack pointer and **`d1`** as an assembly temporary register.

3.2 Changes to MGEN to get DGEN

After studying how MGEN works, we identified major changes that were needed to

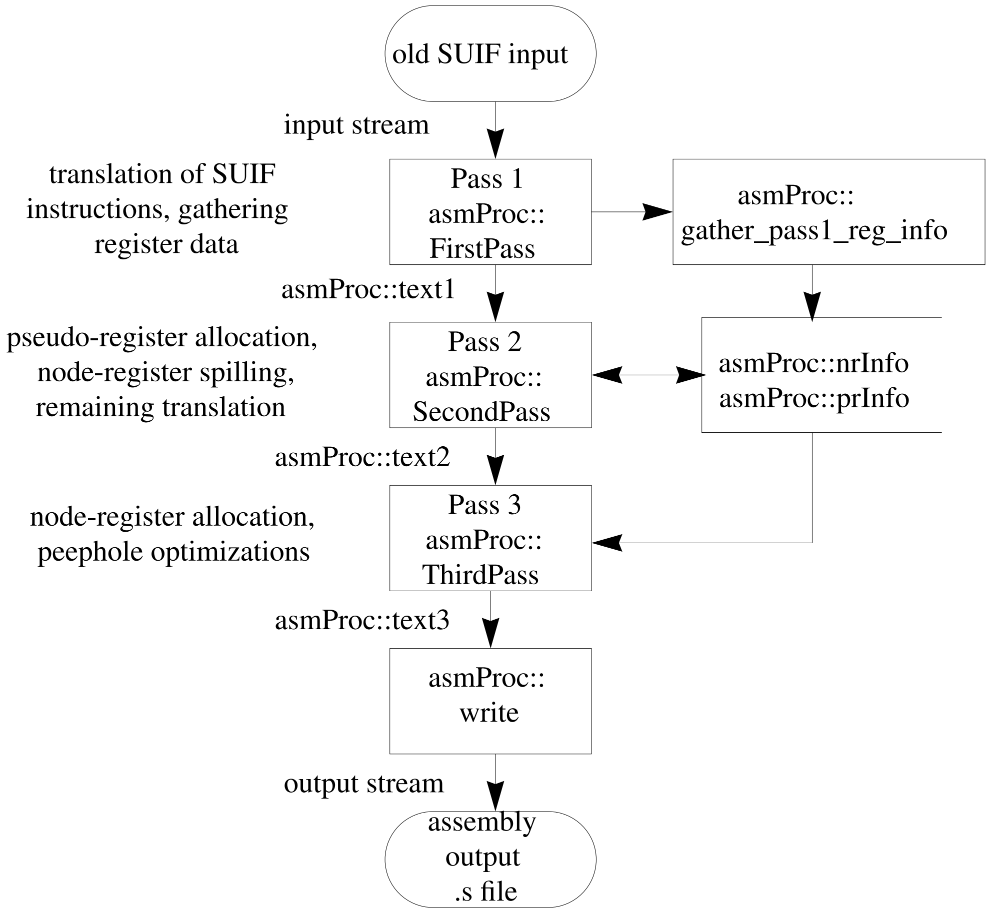

Figure 3.2 MGEN Structure

transform MGEN into DGEN, a code generator for our DSP architecture. This required identifying major differences between the MIPS R3000/R4000 architecture[Hei94], and the model DSP architecture [Sin92][Sag93]. These differences can be summarized as follows:

- The DSP architecture has an address register file that is not present in the MIPS architecture.

- Similarly the DSP architecture has instructions to manipulate the address registers.

- The DSP architecture does not support the direct addressing mode that MIPS supports.

- The DSP architecture does not support function calls through a register .

- The DSP architecture has four -address instructions which are typically useful in DSP applications such as **`madd`** or **`incmod`**.

- Because the DSP architecture has an address register file, the register usage conventions on the DSP are different from MIPS.

- The DSP architecture does not support the **`double`** floating point type that MIPS supports.

- The DSP architecture does not have the byte and double-byte variations of the integer instructions that MIPS has.

- The DSP architecture has an address register **`a0`** which is hardwired to the value 4 and which is normally used by the compiler to increment or decrement a word pointer.

What follows next is the description of the changes made to the MGEN code to transform it into DGEN. Figure 3.3 shows a schematic block diagram of DGEN. As in Figure 3.2, ovals

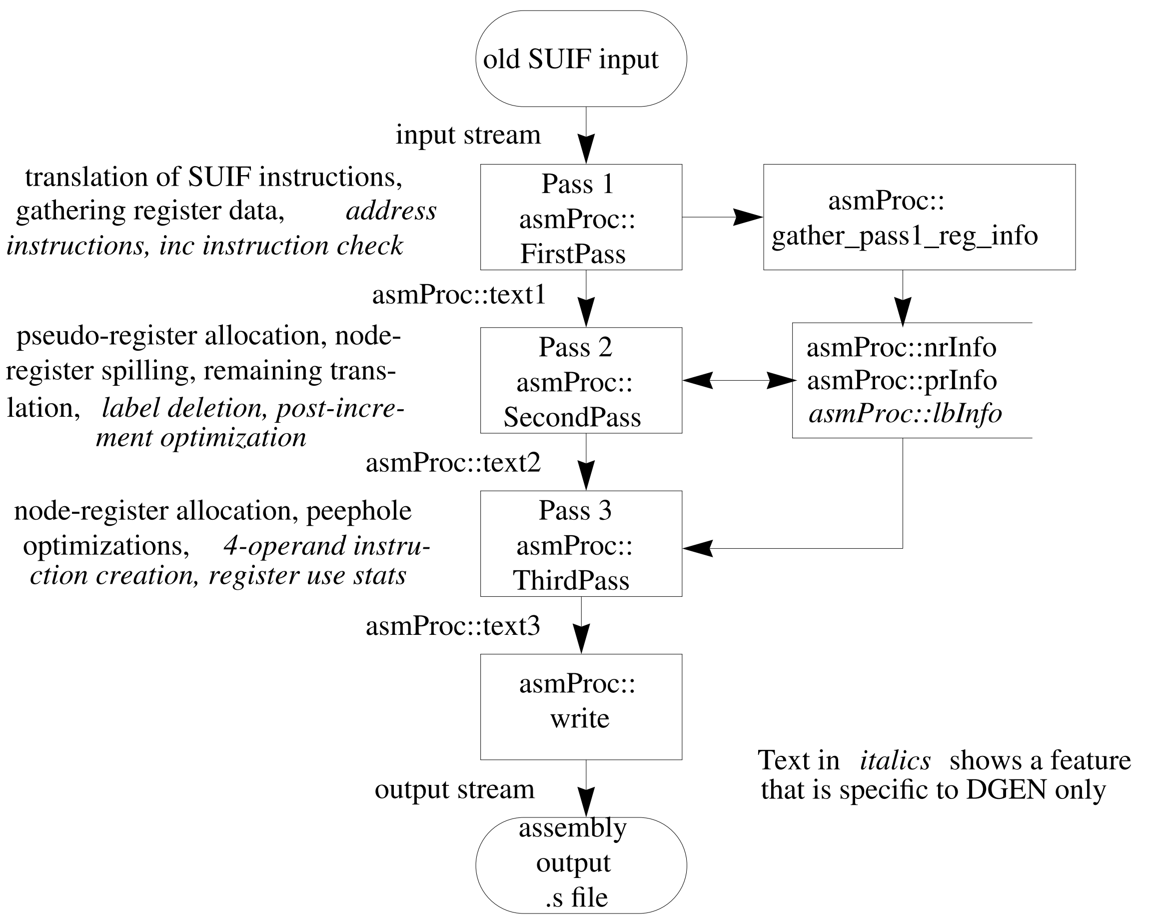

Figure 3.3 DGEN Structure

represent input/ output files, boxes represent functions or methods and the three-sided open box represents a data-structure in memory.

The first major change that was identified and carried out was the new register type for the DSP architecture. To define the address register type, the **`ai_reg`** class and a related supporting **`ai_reg_types`** enum type were modified. In addition, two classes **`ai_reg_src`** and **`ai_reg_dst`** derived from **`ai_reg`** were added, so that a register object while constructing itself sets its own type based on the op-code of the instruction which is being constructed.

Next the instruction formats for the instruction set of the DSP architecture had to be incorporated into the **`asmInstr`** class hierarchy. Appendix A shows the class layouts related to all the instruction formats that were defined for the DSP instruction set. As an example, the class **`ai_r4`**, which is associated with the format **`aif_r4`**, was created to represent the four-operand instructions of the DSP-processor. The ten constructors of this class encapsulate most of the information or logic that is required to build the four-operand instructions. Assembly language op-codes, which are closely associated with the instruction formats, were modified in the **`ai_ops`** enum type and classified according to their instructions' formats. The corresponding text mnemonics, which are used by the output function to write assembly instructions into a text file, were also modified in the **`aoNames`** array.

After modifying the register and instruction class definitions, the next major change involved the three passes.

Pass one is concerned mainly with translating SUIF instructions into DSP machine instructions. Various translation routines (with names like **`asmProc::Translate`*`XX`*`Op`**) required modifications. Some of the major changes in pass one are described below in the order of their importance:

- Since SUIF format allowed instructions with mixed integer and address operands, the pass one translation routines were modified to generate register moves that essentially perform type conversion before emitting the actual translated instruction whenever necessary. See Section 3.3.2 for a detailed description of this change.

- The translation routines now generate address type instructions for address operands.

- Pass one treats all double float type operands and declarations as if they were single float type because the DSP architecture doesn't support double float type. See Section 3.3.1 for a detailed description of this change.

- MGEN maintains a *hold* list of instructions that load constants or immediate operands. It looks up this *hold* list to determine if it can generate immediate operand instructions. Because the DSP does not support instructions with direct addressing mode or with immediate operands, DGEN does not need to maintain the *hold* list to the same extent. For example, there is nothing equivalent to the **`ADDI`** (add immediate) instruction of MIPS [Hei94] in the DSP.

- On the other hand DGEN needs to maintain the *hold* list for those instructions which load function names or addresses into node or pseudo registers. This change was necessary because the DSP does not support function calls through registers, which means a function call through a pointer is not supported unless it is possible to copy propagate, using the *hold* list, the name of the function through the pointer to the place of the call. When DGEN is unable to do this copy propagation, it terminates with an error message.

- In the case of the **`add`** instruction of address type, DGEN has to ensure that the first operand is an address of some object in the memory and the second operand is the increment. See Section 3.3.3 for a detailed description of this requirement and its implementation.

- In DGEN, the type of a **`store`** instruction is determined from the type of the second operand, **`src2`**, as opposed to the type of the first operand, **`src1`**, as is done in MGEN. This change was made necessary by the reuse feature of the induction variable creation optimization which is described in Chapter 4.

Pass two consists of register allocation or stack allocation of pseudo-registers and, if necessary, spilling of node registers. It also does some amount of translation that was not done in pass one and minor peephole optimizations. Some of the major changes in pass two are described below:

- Because the DSP architecture does not support direct addressing as the MIPS processor does [Hei94], the routines that generate stack load or stack store (e.g. **`asmProc::store_to_stack`**) instructions were changed to generate instructions that the DSP architecture supports.

- Because the DSP architecture does not support byte and two-byte variations of integer instructions, the routine **`asmProc::expand_struct_move`**, which generates code for aggregate moves, was changed to make use of the DSP architecture's full word instructions.

- Pass two of MGEN checks that there are no live node registers at the end of a procedure or there are no uses of undefined node registers at the end of a procedure. The reason for this is that each node register is defined exactly once and used exactly once, so the number of uses should equal the number of definitions for all categories of node registers. In case of a discrepancy, MGEN generates an error and terminates. However this check was relaxed in DGEN, so that in case of live node registers (i.e. more node register definitions than uses), it would just generate a warning and continue processing. This was done because of the change that was made with respect to the copy propagation of function name operands through pseudo and node registers.

- Pass two implements an optimization related to the deletion of unused labels. This is described in detail later in Section 3.3.1.

- Pass two implements an optimization related to the post-increment and post-decrement operators. This is described later in Section 3.4.2.

- In DGEN a pseudo-register is allocated a hard register from the pool of 'saved-registers' even if its usage-count (present in the **`asmProc::prInfo`** table) is less than or equal to two. The original designer of MGEN decided not to assign hard registers to pseudo-registers whose usage-count was less than or equal to two, with the reasoning that a pseudo-register's usage-count has to be more than the number of instructions it takes to save and restore a machine register. However the usage-count as calculated in pass one does not take into account uses of a pseudo-register inside a loop, so this restriction was removed in DGEN.

- MGEN converts calls to built-in functions into system interrupts. In case of the DSP development environment, this is done by the post-optimizer. So the built-in function support has been removed from DGEN.

- For some unknown reason MGEN used to assign a stack location to a pseudo-register even if the latter is allocated a hard register. As a space saving optimization, DGEN does not assign a stack location to a pseudo-register in such cases.

Pass three consists of register allocation for node registers, creation of procedure header and trailer instructions and some final peephole optimizations. Some of the major changes made to this pass were:

- Because of the addition of the address register file, the code for the temporary register allocator function **`asmProc::next_free_TR`** was augmented to include allocation of address registers.

- Because the DSP does not support direct addressing mode, this pass uses address register **`a1`** as a running address and **`a0`** for increment/ decrement in the instructions that it generates within the function header and trailer to save and restore registers.

- Similar to pass two (see page 15), this pass now generates warnings (instead of error and termination) for unused live node registers.

- This pass uses register **`a0`** whenever a value of four is used in an address instruction.

- This pass looks for opportunities to replace a pair of instructions with one of the DSP's four operand instructions. This optimization is described in detail in Section 3.4.3.

- If the function being compiled is **`main`**, this pass will not create the function header and trailer which contain the code for saving and restoring the pool of 'saved-registers' of the processor respectively . This can be considered as an optimization.

- This pass now also prints statistics on the usage of temporary registers, registers which are assigned to node register . These statistics can be used to fine-tune the partitioning of available machine registers into 'saved-registers' and 'temporary-registers'.

- Some diagnostics were added to indicate the numbers of node registers that were defined but not used.

3.3 DGEN Enhancements

This section describes DGEN enhancements that cannot be characterized as optimizations, but rather changes that are either necessary or very useful for the DSP architecture. Although some of these enhancements and the optimizations described in the next section might already have been covered in the previous sections where changes made to MGEN were described in some detail, this section might be useful for readers who would like an overall and general description of these changes.

3.3.1 Optional Conversion of Double to Float

Since the DSP architecture does not support the double precision float type (the C **`double`** type), DGEN converts all double type declarations, references and operations to single precision float type (the plain **`float`** of C) as the default behavior . However in view of any future requirements for double float type support, DGEN has been set up so that it is possible to preserve the double type (i.e. generate it in all declarations, references and operations) with the proper setting of a boolean variable called **`doubleAsSingle`**.

When **`doubleAsSingle`** is set to FALSE so as to generate the **`double`** type in the output code, the generated code will have the following properties:

- In declarations, the keyword **`.double`** will be used to denote the type **`double`**.

- The suffix **`.x`** will be used to denote instructions of type **`double`**, just like the suffix **`.s`** is used to denote instructions of type single **`float`**.

Operands on the other hand, remain unchanged. In other words, instruction operands are still the single-precision, float-type registers, **`f0`** to **`f31`**.

3.3.2 Address and int type issues

Almost all of the enhancements in this category are necessitated by a particular aspect of the target DSP architecture: the presence of an address register set and the address type instructions.

When the SUIF compiler was first developed at Stanford University, it was targeted only to the MIPS processor which does not distinguish between address and integer operands. Not surprisingly, the SUIF intermediate format reflects the nature of MIPS architecture as far as address and integer operands are concerned and the SUIF code, which is input to MGEN or DGEN, combines the use of address and integer operands in instructions. In contrast the SUIF code avoids mixing the use of integer and float operands and uses the **`cvt`** instruction instead to convert float operands into integer temporaries and vice-versa thus enforcing strict type-separation between integer and float. This is consistent with the MIPS architecture[Hei94], where both address and integer operands are stored in the same register set and are manipulated with the same set of instructions, so that there is no need for a **`cvt`** instruction when an integer operand is to be used with an address operand (or vice-versa) in an instruction.

On the other hand the DSP architecture dictates that address operands be stored in address registers and manipulated with address instructions. If an integer operand is used in an address instruction, then that operand needs to be moved to a temporary register of address type before being used in the address instruction which is equivalent to a **`cvt`** instruction. But since SUIF does not always generate this **`cvt`** instruction, extra checks were added to pass one of DGEN to initiate the required register move instructions (**`mova2d`** and **`movd2a`**).

The logic that was added to take care of these problems can be described in the following steps:

- If a SUIF instruction is of address type, as determined by the type of its destination, and it has an equivalent address type instruction in the DSP architecture (e.g. **`add`**'s equivalent is **`inc`**), then the logic verifies that both of the source operands of that instruction are of address type. If either of the operands is not (i.e. that operand is of integer type), then it generates a **`movd2a`** instruction to move that operand to an address type register.

- On the other hand, if a SUIF instruction is of integer type, or if it is of address type but has no equivalent address type instruction on the DSP (e.g. **`mul`**), then it verifies that both of the source operands of that instruction are of integer type, and if either of them is not (i.e. that operand is of address type), then it generates a **`mova2d`** instruction to move that operand to an integer type register.

- Finally, if an address type instruction was changed to integer type because of its non-availability in the DSP's address instruction set, then the logic creates a **`movd2a`** instruction to copy the temporary register that was the destination of the instruction to the real address type register of the original instruction.

3.3.3 Correct operand order in the `inc` instruction

The reason for this enhancement is to support modulo addressing, as it is implemented in the target DSP instruction set, the post-optimizer ***OptimizeDSP*** and the DSP simulator ***dspsim***. (see Section 4.3.1 of [Sag93]). Whenever the modulo addressing option is enabled, the post-optimizer looks at each **`inc`** and **`dec`** instruction, and tries to determine if its destination is used to address an element of a known array; and then it converts this instruction to an **`incmod`** or **`decmod`** instruction with the size of the array as the third operand which is the modulo value. However for this conversion to work properly, it is necessary that the first operand of the **`inc`** or **`dec`** instruction be pointing to the array and the second operand be the offset of the required element or the increment for the next element in the array; and not the other way round. This requires that the original **`inc`** instruction, and hence the original SUIF **`add`** instruction, also should have its operands placed this way. Note that this problem does not arise in case of **`sub`** or **`dec`** instructions, because these are not commutative.

However a SUIF **`add`** instruction may not always have its operands placed this way. To solve this problem, the following logic was added to pass one of DGEN: If the second operand of an **`add`** instruction, and its destination are the same, then swap the two source operands, so that the destination and the first source operand of the **`add`** instruction now match. In other words, the first source operand is now a pointer into the array and the second source operand is the offset or increment for that array.

3.4 DGEN Optimizations

This section describes all the optimizations present in DGEN. These are described in the following sub-sections in the order of their effectiveness in terms of the performance improvement in the output code, except that the last sub-section describes optimizations that have been copied from MGEN. The ones that were added to DGEN over the course of this project, as and when opportunities were found to implement these optimizations, form the bulk of these optimizations.

A number of these optimizations were found to be necessary in our initial testing of the SUIF compiler and DGEN, when we compared the SUIF output code with the GCC output code and discovered some obvious opportunities for optimizations in DGEN. Thus it can be said that these optimizations are essentially those which the GCC compiler already performs in its back end.

3.4.1 Unused labels optimization

One of the most important features of the DSP architecture is the Very Long Instruction Word (VLIW) which enables multiple operations to be executed concurrently in the multiple functional units. This is how parallelism is used in this architecture. However both of our front-end compilers (SUIF and GCC) produce only sequential code, which is then compacted by the post-optimizer to produce code consisting of multi-operation VLIW instructions.

Since instruction compaction cannot change the control flow of a program, the post-optimizer can compact operations only within a basic block. The post-optimizer, by doing data-flow and dependence analysis, tries to find a maximal set of independent and orthogonal operations within a basic block that it can compact into a long instruction for the available functional units of each kind. Thus the bigger a basic block, the more opportunities there are for the post-optimizer to compact operations into instructions.

In our initial testing of the SUIF code, we found poor compaction of its code by the post-optimizer and hence poor performance due to low levels of parallelism. When the SUIF output code was analyzed, it was found that SUIF generates many spurious labels in the code, even if there are no jumps to those labels. The reason for this is that, when high SUIF code is converted to low SUIF (dismantling of syntax trees into flat instruction lists), the expander generates all the labels that are present in a node and there is no other phase that removes the unused labels. For example, when a **`for`** loop is dismantled, it generates both the break label (the target of a jump generated from a **`break`** statement), and the continue label (the target of a jump generated from a **`continue`** statement), even if there are no **`break`** or **`continue`** statements in the loop and hence no jumps to these labels. Ideally an optimizer or a data flow analyzer should ignore unused labels while splitting the code into basic blocks. However our post-optimizer , while creating the control flow graph, treats every label in the code as the beginning of a new basic block. Hence the unused or spurious labels in the SUIF code resulted in more basic blocks than necessary , and hence poor compaction as compared to the compaction results of the GCC code which was found to be free of unused labels. See Chapter 6 for a comparison of the performance of the code that has unused labels with the code that does not have them.

The optimization that needed to be implemented in DGEN is fairly simple. A dictionary or cross-reference table was created for labels in the **`asmProc`** class (**`asmProc::lbInfo`**). This table (indexed by an integer , since this table contains entries only for SUIF generated labels) stores the number of references to a label. This table is updated in pass one to record all references to labels in the code. In pass two, a label is removed if the number of references to it is found to be zero.

This optimization is an example of how seemingly unimportant or unobvious optimizations can significantly improve the performance of the generated code when the compiler is able to utilize the special features of the target architecture effectively . In this case, the code generator of the compiler , by deleting unused labels, was able to generate code with bigger basic blocks that resulted in better compaction for the target VLIW architecture.

3.4.2 Optimization of Postfix operator code

The C post-increment and post-decrement operators are not very well handled by the SUIF front-end, basically because the SUIF expression trees cannot represent the side effects of the post-fix **`++`** or **`--`** operators. For example, the SUIF compiler will translate the following code fragment shown on the left hand side into code shown on the right hand side:

```
a = b++ * c;        ==========>        suif_tmp1 = b;
                                       b = suif_tmp1 + 1;
                                       a = suif_tmp1 * c;
```

Figure 3.4 SUIF post-increment code

The temporary **`suif_tmp1`** is created to preserve the old value of **`b`** and at the same time to schedule the post-increment side effect of incrementing the value of **`b`** by one. This code that SUIF generates has a number of drawbacks:

- For every use of the post-increment or post-decrement operator the compiler creates a unique variable symbol (**`suif_tmp1`** in the above case) that it uses in the expressions to schedule the side-effect. This combined with poor register allocation gives rise to increased register pressure.

- This code is not quite efficient as compared to the code that GCC generates which would appear as (for the code fragment on the left hand side of Figure 3.4):

  ```
  a = b * c;
  b = b + 1;
  ```

  Figure 3.5 GCC post-increment code

  This has only two instructions as compared with the three that SUIF generates.

- The true dependency created by first assigning a value to **`suif_tmp1`** and then using it in the second and third instructions requires at least two long instructions, even if operation slots are available for compaction. Whereas the two instruction code that GCC generates has only an anti-dependency in **b** that still allows both operations to be executed concurrently in the same long instruction.

Because of this, we decided to modify the post-fix code that SUIF generates. The possibilities of modifying the SUIF front-end (called **snoot**) or some of the other intermediate SUIF passes were investigated. However we decided to perform this optimization in DGEN, since we wanted to minimize changes to other passes of SUIF to maintain compatibility and since all the information that is required to perform this optimization is available in DGEN.

This optimization is fairly straight-forward and is implemented in pass two. Whenever a pseudo-register associated with a variable-name of the format **suif_tmp***nn*, where *nn* are decimal digits, is used in the code exactly as shown above in Figure 3.4 in a sequence of three instructions, this sequence of instructions is modified so that it appears like the two-instruction sequence as shown in Figure 3.5.

3.4.3 Creation of four-operand instructions

This optimization is necessitated by the availability of four operand instructions in the DSP architecture and to keep up with the GCC compiler, since the GCC compiler does it! The DSP architecture has six four-operand instructions that are very useful in DSP application code. For example, "**madd.d di,dj,dk,dl**" is a four-operand instruction which executes the multiply and accumulate operation.

This optimization has been implemented in pass three, where the instruction stream is checked for two consecutive instructions which can be folded into a single four-operand instruction. For example a **mult.d** instruction and an **add.d** instruction can be combined into an **madd.d** instruction provided that the destination of the first instruction is a node register which is used as one of the source operands by the second instruction.

3.4.4 Function header and trailer elimination for main

MGEN and DGEN generate code in which they use a calling convention whereby the callee saves and restores the machine registers on function entry and exit respectively. Since the **main** function is not called by any other function, the function **main** does not need to have the function header and trailer which are nothing but the register save and restore code. Thus, in pass three, DGEN checks if the current function being compiled is **main**, and if it is, it does not generate the function header and trailer.

3.4.5 Other miscellaneous optimizations

This section describes other minor optimizations added to DGEN, and they are as follows:

- DGEN uses address register **a0**, whenever the value of four is required in an address instruction, or if an address register is being incremented or decremented by four.

- Whenever function parameters are passed through the parameter registers, DGEN avoids copying them to the stack (as opposed to MGEN which passes parameters through both the parameter registers and the stack).

- Addition or subtraction of zero to an operand is eliminated.

- DGEN does a limited amount of copy propagation of immediate values which are function names, through pseudo and node registers to avoid generating a function call through a register since the DSP does not support it.

3.4.6 Optimizations copied from MGEN

The optimizations described here are the original MGEN optimizations which have been preserved in DGEN:

- If a memory load into a node register is followed by a register copy from that node register , then these two instructions can be collapsed into a single memory load into the final destination register . Similarly a register copy into a node register followed by a memory store from that node register can be collapsed into a single store. Recall that a node register is defined only once and used only once and that is why this optimization on the node register is possible. This optimization is also extended to include all register copy instructions defining a node register that are followed by other operations using that node register , so that the register copy can be eliminated by the appropriate substitution of the node register by the source of the copy instruction.

- Register **`d0`** is used in place of value zero.

- Useless moves such as " **`mov.d dx,dx`**" are deleted, although such instructions are rarely encountered.

3.5 DGEN's Code Output: Some Observations

After implementing most of the optimizations described above, the performance of the code generated by DGEN was analyzed. W e observed that many of the DGEN optimizations were very effective in improving the performance of the SUIF code (see Chapter 6, which discusses the performance impact of a couple of optimizations described in this chapter). However the code produced by SUIF was still not very efficient as compared to the code produced by GCC, because of a lack of higher level scalar optimizations such as induction variable creation and loop-invariant code motion. When we used the scalar optimizer that comes with SUIF , we found that it was not sufficient or stable enough to serve our needs. As a result, we ended up in a situation where the SUIF code was still significantly slower than the GCC code.

In the next chapter we explain how we improved the quality of SUIF-generated code by implementing several machine-independent scalar optimizations.

Chapter 4 Optimizations

This chapter presents the optimizations that were implemented in the SUIF compiler to overcome the limitations imposed by our GCC-based compiler while not sacrificing performance. This effort at implementing new optimizations in the SUIF compiler was motivated by two main factors: the performance of the *C-style* benchmarks and the results of using SUIF's existing scalar optimizer. So the initial section of this chapter presents some background information on the results of performance analysis as well as SUIF's existing support for some of the scalar optimizations that we wanted. Later on we describe each of the optimization passes we implemented in detail. These optimizations are: induction variable creation, dependence analysis and common array sub-expression elimination.

4.1 Preliminaries

Our objective was to have a retargeted SUIF compiler that would be able to generate code for the *C-style* benchmarks whose performance would be comparable to that of the original version of those benchmarks. Because they were written in the 'DSP-style' and were software-pipelined, we call the original version of the benchmarks *DSP-style with SWP*, where SWP stands for 'software-pipelined'. What we mean by 'DSP-style' is discussed in detail in Section 2.3. This section briefly describes the *C-style* benchmarks, the outcome of our performance analysis, as well as our attempts at using SUIF's existing scalar optimizer which would enable the reader to understand why we decided to write our own optimization passes for the SUIF compiler.

4.1.1 *C-style* benchmarks

As stated earlier in Chapter 2, all of the previous work that was done on the DSP architecture and the GCC-based compiler was done using the benchmarks that were written in the 'DSP style'. See Appendix C of [Sin92] and Appendix B of [Sag93] for a listing of these benchmarks. Since we wanted to do away with the restrictions of the 'DSP-style' with our new retargeted SUIF compiler, it was decided to modify the *DSP-style with SWP* benchmarks into semantically equivalent benchmarks that would be free of the restrictions of the 'DSP-style' and would look like natural C code. The task of modifying the *DSP-style with SWP* benchmarks to get the new benchmarks, which we call *C-style* benchmarks, is a form of reverse engineering, since it involved undoing the constraints of the 'DSP-style' to get a true high-level language program. The main steps that constitute this undoing of restrictions of the 'DSP-style' are:

- using array index notation instead of pointer notation everywhere,

- removal of the programmer-introduced software pipelining in `for` loops and the resulting elimination of temporary variables,

- using multi-dimensional arrays wherever appropriate instead of the single dimensional arrays that the DSP-style requires,

- using the `for` loop index variable wherever required and appropriate instead of a specially created induction variable that the DSP-style forces, and

- undoing other manual optimizations such as putting loop-invariant computations back inside the loop.

4.1.2 Performance of the *C-style* benchmarks using SUIF

After we developed an initial version of the DSP code generator DGEN without the optimizations described in Chapter 3, the code it generated for the *DSP-style with SWP* benchmarks as well as for the *C-style* benchmarks was tested and compared for performance with the code generated by the GCC-based compiler. Although Chapter 6 describes in detail the experimental framework and the results of this performance analysis, we will briefly summarize here those results of the performance analysis that are relevant to the discussion of scalar optimizations and our decision to develop our own scalar optimizer. Figure 6.4 shows the initial results that we obtained with the new SUIF compiler. You can see a large difference between the performance of the code generated by the GCC-based compiler and code generated by the SUIF compiler; the latter being slower in almost all cases. This was observed in the *DSP-style with SWP* benchmarks as well as the *C-style* benchmarks. Even with the SUIF compiler's optimization switch turned on, it was observed that the code generated by the SUIF compiler was slower than the code generated by the GCC-based compiler by an average factor of 2.0. This can be seen in the graph in Figure 6.9.

When the SUIF-generated code was analyzed, it was found to be inefficient in many respects. Some of these inefficiencies could be fixed in DGEN and these optimizations have been described already in Section 3.4. When these optimizations were implemented, we observed significant improvement in the performance of the SUIF code for both the *DSP-style with SWP* as well as the *C-style* benchmarks as can be seen in the graphs in Figure 6.7 and Figure 6.8.

However our ultimate objective was to have a compiler that would not impose the constraints of the 'DSP-style' on the source programs like the GCC-based compiler did, but would deliver similar performance. As you can see in the graph in Figure 6.4, there is a large difference in the performance of the code for the *DSP-style with SWP* benchmarks and the code for the *C-style* benchmarks, whether we use the GCC-based compiler or the SUIF compiler. This suggests that the programming style has greater impact on performance than the compiler, especially when we are using a specialized architecture such as the DSP architecture. Our goal was to have a compiler that would take into account the features of the target architecture and perform suitable program transformations that result in performance comparable to that produced by the DSP programming style. In other words, the SUIF compiler should ideally generate code for the *C-style* benchmarks that would run about as fast as the code generated by the GCC-based compiler for the *DSP-style with SWP* benchmarks. As we will see in the next sub-section, this requires the use of some powerful scalar optimizations in the SUIF compiler.

4.1.3 Need for scalar optimizations

If you look at the constraints imposed by the DSP-style, they clearly point to a lack of some scalar optimizations in the GCC-based compiler. Optimizations missing in GCC are:

i) induction variable creation: This involves replacing linear expressions of the loop index variable with an induction variable. Induction variable in this context means a variable whose value at any time inside the loop can be expressed as a linear expression of the loop index variable (see Section 10.7 in [ASU88]). This optimization would allow the use of array notation inside loops as described in Section 2.3, as well as the use of multi-dimensional arrays. For example, consider the code fragment written partly in the 'DSP-style' in Figure 4.1(a). Its equivalent code in the natural style is shown in Figure 4.1(b).

```
i1 = 0; pa1 = a;
for (i = 0; i < N; i++) {
  pa = pa1;
  j1 = 0;
  j2 = 0;
  pb = b;
  for (j = 0; j < i; j++) {
     *pa++ = i1 + j1;
     *pb++ = j2;
     j1 = j1 + 2;
     j2 = j2 + 5;
  }
  i1 = i1 + 3;
  pa1 = pa1 + M;
}
```

(a)

```
for (i = 0; i < N; i++) {
  for (j = 0; j < i; j++) {
    a[i][j] = i * 3 + j * 2;
    b[j] = 5 * j;
  }
}
```

(b)

Figure 4.1 Effect of induction variable creation on DSP-style

The addresses of array elements in Figure 4.1(b) are linear expressions of the loop indices, and hence these expressions can be converted into induction variables that will result in the same code produced from the code in Figure 4.1(a). Thus, the linear address expression `b + 4 * j` which results from the array access `b[j]` corresponds to the induction variable `pb` which appears in Figure 4.1(a). This optimization would also allow the programmer to use the loop index variable in linear expressions inside the loop. For example, in the above code fragment the linear integer expression `5 * j` that appears in Figure 4.1(b) corresponds to the use of variable `j2` in Figure 4.1(a), where `j2` is an induction variable.

ii) loop-invariant code motion: Any code inside a loop that computes the same value in all iterations of the loop is called loop-invariant computation. As an optimization, this loop-invariant computation can be moved out of the loop to a *loop preheader* where it can be executed only once before control enters the loop body. Because of this optimization, the application programmer does not have to worry about manually doing this optimization in the code. As an example, in Figure 4.1(b) the expression `i * 3` is loop-invariant with respect to the inner `for` loop, since `i` does not change during the iterations of the inner `for` loop. If we perform loop-invariant code motion, the expression `i * 3` can be moved out of the inner loop and assigned to a variable. The value of this variable then represents the value of the original expression when control is inside the inner loop. The DSP-style of Figure 4.1(a) essentially achieves this by moving `i * 3` out of the inner loop and assigning it to the variable `i1`.

iii) software pipelining: this involves rearranging loop iterations so as to exploit instruction level parallelism opportunities across adjacent iterations of the loop. Although this is not exactly a scalar optimization, the back-end of the compiler could perform this optimization which is actually a sophisticated form of instruction scheduling. This would eliminate the need for manual software pipelining or loop adjustment by the application programmer. As an example, Figure 4.2(a), shows a code fragment written in the 'DSP-style', that achieves the same effect as applying the software pipelining optimization on the code shown in Figure 4.2(b) which is written in the natural style.

Since overcoming the constraints of the DSP-style required the use of these scalar optimizations, we explored the possibility of using SUIF's own scalar optimizer.

```
pa = a;
pb = b;
temp = *pb++;
for (i = 1; i < N; i++) {
  *pa++ = temp;
  temp = *pb++;
}
*pa = temp;
```

(a)

```
for (i = 0; i < N; i++) {
    a[i] = b[i];
}
```

(b)

Figure 4.2 Effect of software-pipelining

4.1.4 Using SUIF's own scalar optimizer

The SUIF compiler has its own scalar optimizer called ***oynk***, which uses the old SUIF format [SUL94][SUC94]. When we attempted to use ***oynk***, we faced a number of problems. First of all, a number of benchmarks did not run successfully, probably as a result of bugs present in ***oynk***. The presence of bugs has been documented in the SUIF release notes [SUC94]. Secondly, ***oynk*** was developed as part of the old SUIF compiler and the SUIF development group at Stanford does not want to support it anymore, and they could not promise the availability of a similar scalar optimizer for new SUIF. It should be noted that there is new SUIF program called ***porky*** which does assorted code transformations which the SUIF development group wanted for their own use. However it does not have the optimizations that we want in our compiler. A related problem of ***oynk*** being an old SUIF program is that when ***oynk*** is used in the compilation it requires both the ***oldsuif*** and ***newsuif*** phases to convert from new SUIF to old SUIF and vice-versa. These conversions result in mismatches between new SUIF and old SUIF, causing the SUIF compiler to terminate without completing the compilation. Another problem we found in ***oynk*** is that it did not find all the opportunities for induction variable creation that we had expected. For example, it failed to apply these optimizations for some nested loops, where an expression in the inner loop is a linear expression of the loop index variables of both of its enclosing loops. Finally, making our own changes to ***oynk*** was not a feasible idea, because it used the unsupported old SUIF format and there was not enough documentation available on either ***oynk*** or the old SUIF format.

As a result of these problems, we decided to develop our own optimizer that would perform exactly those optimizations that we wanted in the SUIF compiler. We call this phase INDVAR and describe it in the next section.

4.2 INDVAR: New optimizer for the SUIF compiler

The previous section described the need for writing a new scalar optimizer for SUIF that would implement the optimizations that we wanted. This section presents the requirement specifications, design decisions and main highlights of this optimizer, which we call *indvar* because of the main optimization it does: induction variable creation.

4.2.1 Requirements for INDVAR

After we decided to write our own optimizer for the SUIF compiler, our first task was to identify the exact requirements for it. The requirements that we identified can be summarized in the following points:

i) The most important optimization it should do is induction variable creation (IVC). See the bullet numbered (i) in Section 4.1.3 which explains the requirement for this optimization.

ii) For the IVC optimization to be very effective, it should also detect loop-invariant expressions, and perform code-motion for these expressions. As an example, if you look at the code fragment in Figure 4.1(b), the array access **`a[i][j]`** involves the outer loop index variable **`i`** which is a loop-constant as far as the inner loop is concerned. Correct detection of this will enable the address expression for **`a[i][j]`** to be converted into two induction variables, one for the inner loop and the other for the outer loop.

iii) The optimizer should reuse the symbols that it creates for the induction variables as much as possible. This requirement was necessitated because the retargeted SUIF compiler does not have a good register allocator; the register assignment of DGEN does not take into account the live ranges of variables. This means that if you have two **`for`** loops, where neither nests the other, they should share an induction variable to minimize the number of local variables in the procedure.

iv) This optimizer should detect common sub-expressions inside a **`for`** loop for which it can use the same induction variable. This would allow common sub-expressions to be eliminated when these belong to the class of expressions that are optimized by induction variable creation and loop-invariant code motion.

These requirements give us a fairly good idea of what kind of an optimizer we need. Of course, we have not considered the requirement related to the software pipelining aspect of the 'DSP-style'. Since software-pipelining is a sophisticated form of instruction scheduling, the ideal place where it should be implemented is the post-optimizer. Although this is outside the scope of this work, the software pipelining optimization pass requires certain dependence information from the SUIF compiler. Section 4.3 describes how we modified the SUIF compiler to provide the necessary dependence information.

4.2.2 Design Decisions

The INDV AR optimizing phase was designed to work with the new SUIF format. The SUIF representation and the structure of other SUIF optimizing passes give us an indication of what INDV AR should look like. The design of INDV AR was shaped by the structure of a typical SUIF optimizing pass [SUC94], and the requirements for INDV AR which were described in the previous sub-section. Some of the important design decisions that were made during the development of INDV AR are summarized below and are listed in the order of their importance. Note that all references to a loop imply a **`for`** loop.

- The algorithm for traversing an expression tree should start from the leaves of the tree and work towards the root. During the walk it should look at absolute or loop constants, variables (classified as either loop constants, the loop index or others), and instructions such as **`add`**, **`sub`**, **`neg`**, **`cpy`** and **`mul`** that preserve linearity. While going towards the root, it should calculate the multiplier for the expression (i.e. constant of proportionality with respect to the loop index), and create an induction variable when it finds a maximal linear expression. This aspect of the design is related to requirement (i) in Section 4.2.1.

- Consider loop-invariant or loop constant expressions as a special case of linear expressions with a multiplier or increment of zero, so that we will achieve code motion for loop-invariant expressions as part of this optimization. This deals with requirement (ii) in Section 4.2.1.

- Create induction variables only for those expressions that are not trivially simple. Examples of trivially simple expressions are an integer constant or a variable that is either a loop-constant or the loop index variable itself.

- In case of nested loops, analyze and optimize these loops starting from the inner -most loop, and go towards the outer loop. While optimizing a nesting or enclosing loop, look only at the landing pad of the inner loops. A *landing pad* is a place in the `for` loop that SUIF provides, where loop initialization code can be moved. The code in the *landing pad* is executed only once before, and only if, the loop body is executed at least once [SUC94]. This again deals with requirement (ii) in Section 4.2.1.

- While optimizing a `for` loop, maintain a linked list of loop linear expressions for which an induction variable has been created. If an instance of an expression from this linked list is found later in the loop, that instance can be eliminated with the use of this induction variable, thus performing common subexpression elimination for loop linear expressions. This will satisfy requirement (iv) in Section 4.2.1

- Provide a threshold at which to stop creating induction variables. If the total number of variables of a given type in a procedure reach that threshold, INDVAR stops creating induction variables. Since the number of registers is limited in the target DSP architecture, there is no point in creating a large number of induction variables, when we definitely know that at least some of them are going to be spilled to the stack. The spilling of variables, apart from worsening performance, prevents the post-optimizer from successfully performing the modulo-addressing pass.

- Create each induction variable symbol at the function level as opposed to the block level, so that it is available throughout the function and its reuse is possible. See the next bullet for the reuse feature.

- Maintain a linked list of all induction variable symbols already created for the function. Every time a new induction variable is required, try to reuse an existing induction variable symbol which is no longer live. This will satisfy requirement (iii) in Section 4.2.1.

- Have INDVAR run after dismantling arrays but before dismantling anything else [SUC94][SUC94]. This is necessary because INDVAR can work only with `for` loops and the replacement of array index notations with pointers that are induction variables requires dismantling of arrays before INDVAR is run.

- Also have a constant folding pass run after INDVAR, because INDVAR creates initialization expressions in the *landing pad* of the `for` loop. These initialization expressions are created by moving the original linear expressions to the *landing pad* and replacing the loop index variable in them with the lower bound of the `for` loop. The constants in an initialization expression can then be folded into a single constant with a constant-folding pass.

- Provide an option to perform IVC only for address expressions, since we are mainly interested in replacing array index notation with induction variables.

4.2.3 Highlights of INDVAR

INDVAR follows the structure of a typical new SUIF pass. It makes extensive use of functions provided by the SUIF library [SUC94] to process SUIF objects such as files, procedures and tree nodes (syntax nodes) of a procedure. For example, the function

```
process_node(cur_node, data)
{
   if (cur_node is an IF node) {
      call process_node(test clause of the IF node, data);
      call process_node(then clause of the IF node, data);
      call process_node(else clause of the IF node, data);
   }
   else if (cur_node is a LOOP node) {
      call process_node(test clause of the LOOP node, data);
      call process_node(body of the LOOP node, data);
   }
   else if (cur_node is a BLOCK node) {
      call process_node(body of the BLOCK node, data);
   }
   else if (cur_node is a FOR node) {
      call create_ind_vars_in_for(body of the FOR node);
      call process_node(landing pad of the FOR node, data);
   }
   else if (cur_node is a INSTR node AND data is non-NULL) {
      de-reference data to get the params;
      call process_instruction(root instruction, params);
   }
}
```

Figure 4.3 The function `process_node`

`suif_proc_iter` allows the user to specify a function that SUIF should invoke for each procedure that it encounters in the input file. Similarly SUIF provides a method called `tree_node_list::map` that allows the users to traverse a `tree_node_list` with their own function. For more information on these calls, refer to the documentation on the SUIF library [SUC94]. This sub-section broadly describes how INDVAR works by way of pseudo-code and flow-charts. All the function references in the following description indicate functions that have been written by us specifically for INDVAR unless otherwise stated.

INDVAR calls the function `process_proc`, not shown here, for each procedure that is found in the input file. `process_proc` in turn calls the function `process_node`, using the `tree_node_list::map` method, for each node that is present in the node list of that procedure. Figure 4.3 shows the algorithm of `process_node` in the form of pseudo-code. As you can see `process_node` recursively calls itself to traverse the syntax tree, and when it finds a `for` loop, it calls the function `create_ind_vars_in_for` to do induction variable creation.

Figure 4.4 shows the pseudo-code of the function `create_ind_vars_in_for`. This function also calls `process_node` to traverse its node list and passes the address of a special data structure in the parameter `data` of the function `process_node`. Thus `process_node` is also used to process instruction nodes when they are inside a `for` loop

```
create_ind_vars_in_for(for_node, nesting_level)
{
   set up data for induction variable creation;
   for each node in the node list of for_node {
      call process_node(node, data);
   }
   append the increment list to the node list of for_node;
}
```

Figure 4.4 The function `create_ind_vars_in_for`

and when **process_node** encounters an instruction node inside a **for** loop (as indicated by the non-NULL value of the parameter **data**), it calls the function **process_instruction** which traverses the expression tree rooted in the instruction node. We will later see how **process_instruction** traverses the expression tree to create induction variables. When **create_ind_vars_in_for** traverses all the expression trees inside a **for** loop and creates induction variables for that loop, it accumulates a list of instructions for incrementing the induction variables. When **create_ind_vars_in_for** has finished processing the **for** loop, it appends this list to the body of the loop thus generating the iteration increment of the newly created induction variables.

As mentioned earlier, the function **process_instruction** is called to traverse expression trees occurring inside a **for** loop. To understand how this function works, it is necessary to first understand the expression tree structure in SUIF. Figure 4.5 illustrates an expression tree that represents the expression **c + a * b**. The ***add*** instruction has two operands: one is a symbol and the other is another instruction, the ***mul*** instruction. In other words the result of the ***mul*** instruction is used as a source operand by the ***add*** instruction. The dual nature of an instruction operand is encapsulated by the class **operand** of the SUIF library. When a source operand of an instruction *i1* is another instruction *i2*, it means that the result of *i2* is used by *i1* as its source operand. Similarly, when the destination operand of an instruction *i3* is an instruction *i4*, it means that one of the source operands of *i4* is *i3*.

Figure 4.6 shows the flow-chart for the function **process_instruction**. This function processes each operand by calling **process_operand**, which in turn calls **process_instruction** if that operand happens to be an instruction. This way a single **process_instruction** call to the root of the expression tree achieves complete tree traversal. The main task of **process_instruction** is to collect the multipliers for its operands and combine them into one multiplier if possible. A *multiplier* is the constant of proportionality when an operand, whether an instruction or a symbol, is found to be a linear expression of the loop index variable. For a loop-constant expression or operand, the multiplier is the value zero or "zero-multiplier"; for an operand which happens to be the loop index variable, the multiplier is the value one or "unity-multiplier"; and if the expression is not a linear expression of the loop index variable then there is no valid multiplier value or

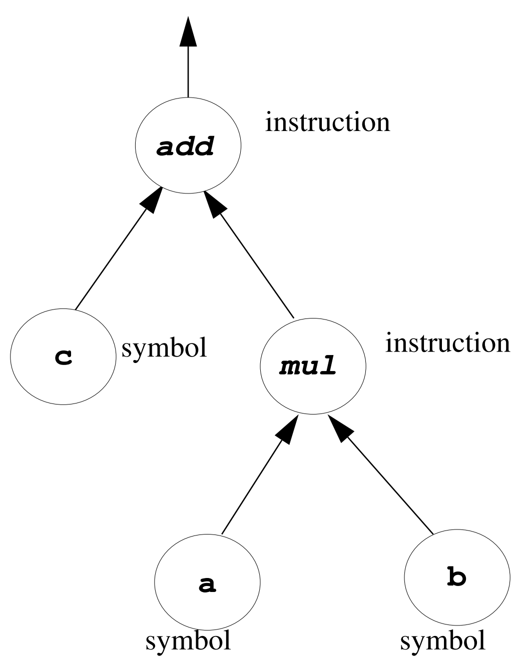

Figure 4.5 An example of an expression tree

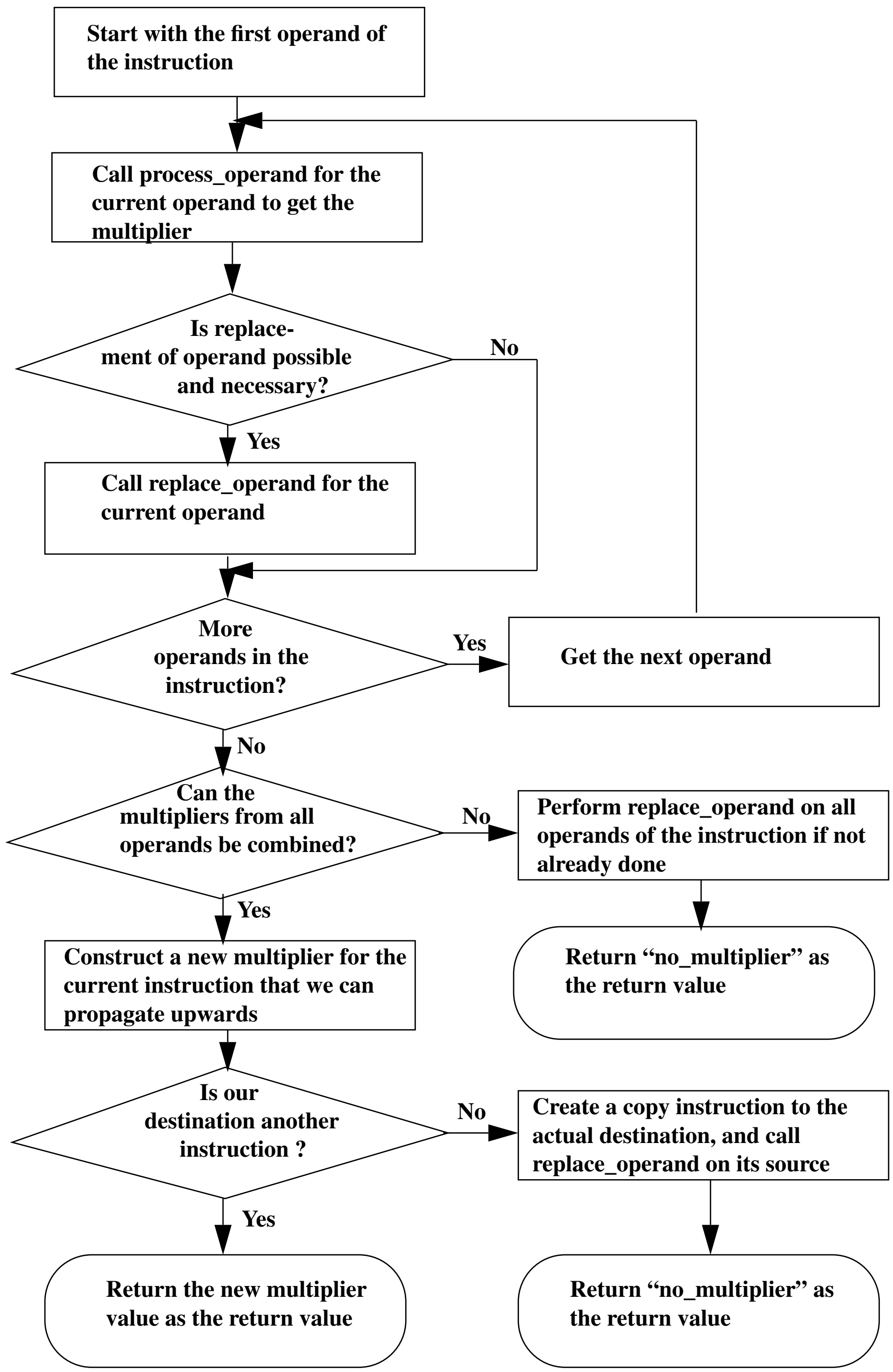

Figure 4.6 Flow-chart of the function `process_instruction`

```
process_operand(cur_operand, data)
{
   if (cur_operand is a variable symbol) {
      var_symbol = variable symbol of cur_operand;
      if (var_symbol is same as current loop's index
              variable) {
         return unity_multiplier;
      }
      else if (var_symbol is a volatile variable) {
         return no_multiplier;
      }
      else if (var_symbol is unchanged in the body of the
                 current for loop) {
         return zero_multiplier;
      }
      else
         return no_multiplier;
   }
   else if (cur_operand is an instruction) {
      instr = instruction of cur_operand;
      return process_instruction(instr, data)
   }
   else {   /* means it is a null operand */
      return zero_multiplier;
   }
}
```

Figure 4.7 The function `process_operand`

the multiplier is said to be "no-multiplier". In all other cases the multiplier itself is typically an expression tree which denotes the value of that multiplier. **`process_instruction`** adds an instruction node to this multiplier expression tree when it determines that all the multipliers of the operands of the instruction it is processing can be combined into a multiplier that can be propagated upwards in the expression tree being traversed. Although replacement of each operand in this case is possible, it is not necessary because the entire expression will be eventually replaced by an induction variable. On the other hand, if the multipliers cannot be combined, such as when the instruction denotes a non-linear operation, it calls the function **`replace_operand`** for each of those operands that has a valid multiplier. In short, the function **`replace_operand`** is called whenever replacement of an operand is possible and also necessary.

For example, consider a node in the expression tree which is an add instruction:

`c = add a, b`

While processing this node, if **`process_instruction`** finds that both **`a`** and **`b`** are linear expressions of the loop index variable, which means they have valid multipliers $\mathbf{m_a}$ and $\mathbf{m_b}$, then the result of the add instruction also has a valid multiplier which is $\mathbf{m_a + m_b}$ and hence **`process_instruction`** can propagate this multiplier upwards by creating an ***add*** node in the multiplier tree to represent the new multiplier for **`c`**. On the other hand, if $\mathbf{m_a}$ is not a valid multiplier, the expression **`a + b`** is not a linear one. In this case, since **`b`** is still a linear expression it can be replaced with an induction variable provided it satisfies other criteria, and **`process_instruction`** would then call the function **`replace_operand`** for operand **`b`**.

Figure 4.7 shows the algorithm for **`process_operand`** as pseudo-code. One thing to note here is that **`process_operand`** is only called by **`process_instruction`**, and its only task is to calculate the appropriate multiplier for the operand, by first determining what kind of an operand it is and then calling an appropriate function for that case. For example, when the operand is a variable which is neither the loop index variable nor a **`volatile`** variable*, it calls a function **`unchanged_by_body`** to determine if the variable is ever modified in the loop body and if it is, then it returns **`no_multiplier`** as the multiplier value, otherwise it returns **`zero_multiplier`** as the return value which means that the variable is a loop constant. Note that the function **`unchanged_by_body`** was copied from the source code of ***porky***.

Finally we will look at the function **`replace_operand`** whose main task is to replace linear expressions with induction variables. Figure 4.8 describes what this function does in the form of a flow-chart. The function **`replace_operand`** is called by **`process_instruction`** when the latter has determined that an operand can be replaced with an induction variable. For example, if **`process_instruction`** is processing the SUIF ***load*** instruction whose source operand happens to be the expression **`k1 + i * k2`** where **`i`** is the loop index variable and **`k1`** and **`k2`** are loop constants, then this expression can be replaced with an induction variable **`ivar1`** that would be initialized in the *landing pad* with the initialization expression **`k1 + i0 * k2`** and incremented at the end of the loop body with the increment **`i1 * k2`**, where **`i0`** is the initial value of **`i`** and **`i1`** is the iteration increment of **`i`**.

A brief explanation of the checks shown in the three diamonds in Figure 4.8 is warranted. These checks are made by **`replace_operand`** before it proceeds to replace a linear expression. The first check ensures that **`cmult`** is a valid multiplier, i.e. it is not a "no-multiplier". A valid multiplier indicates that the corresponding operand is indeed a linear expression of the loop-index variable. The second check ensures that if the value of **`cmult`** is either zero or one, then the operand under consideration is not a trivially simple expression such as a constant or the loop-index variable itself. In these cases, we are not going to improve the performance by either creating an induction variable or doing loop-invariant code motion. The purpose of the third check is to ensure that INDVAR does not create extra variables beyond a certain threshold. Without this check, creating an excessive number of variables would result in increased register pressure and spilling in the code generation phase, that in turn can result in degrading performance.

So far we have seen how induction variable creation is implemented in INDVAR. However there are a few limitations of this optimization that should be taken into account. These are covered in the next sub-section.

4.2.4 INDVAR Limitations

Although INDVAR is very good at performing the optimizations it was designed to perform, it has some limitations. Some of these limitations are inherited from SUIF because of the constraints imposed by the SUIF library and the way SUIF compiler works.

The first limitation is that the INDVAR optimizations work only on **`for`** loops, and that is because a loop index variable exists only for a **`for`** loop as far as the SUIF representation is concerned. It is possible to fix this limitation by writing a new transformation pass that converts a **`while`** loop or a **`do`**-**`while`** loop into a **`for`** loop when it detects the presence of a basic induction variable that can be made into the loop-index variable of the converted **`for`** loop.

Another related limitation is that these optimizations work only on **`for`** loops which are *good* as defined by the SUIF library. For example, a **`for`** loop in which the loop index variable is modified inside the loop body is a *bad* **`for`** loop. It is not possible to fix this

* A variable can be declared as **`volatile`** in C, which means that its value can change in unexpected ways at run-time. This is to tell the optimizer not to use **`volatile`** variables in any data flow analysis or optimizations.

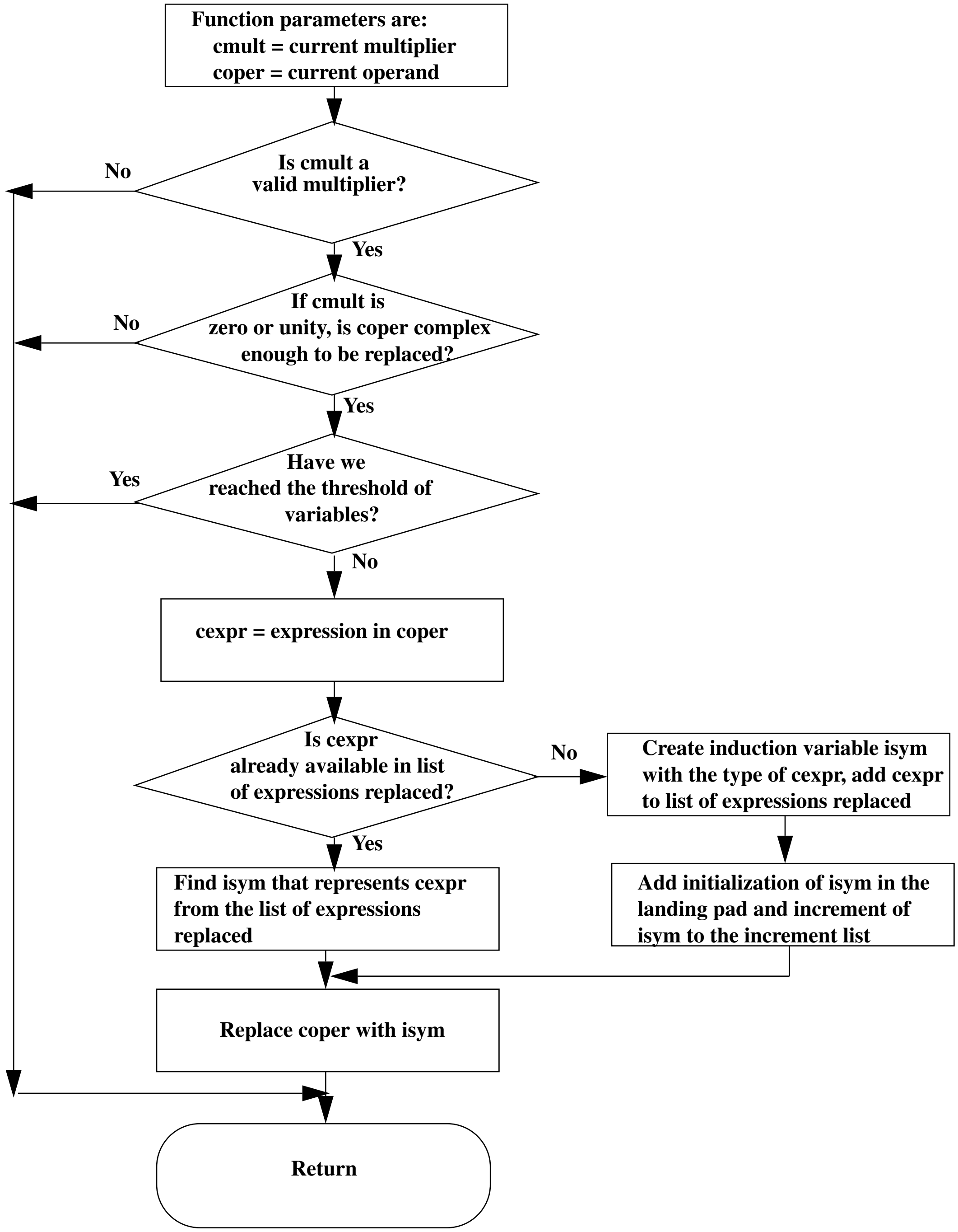

Figure 4.8 Function `replace_operand`

limitation because a *bad* `for` loop essentially indicates that the loop-index variable is not a basic induction variable and it would not be possible to create more induction variables in this loop.

The third limitation is that if a `continue` statement is used inside a `for` loop, the induction variable creation optimization will not work properly. This is because the `continue` statement produces a jump to the end of the loop body which skips the increment instructions inserted by INDVAR. It would be easy to fix this limitation by changing the target of the these jump instructions from the continue label to a label introduced by INDVAR. This label would precede the first increment instruction inserted by INDVAR in the body of the `for` loop thereby ensuring that these increment instructions are executed even when the `continue` statement is executed in an iteration.

4.3 DEP_AN - A Dependence Analyzer for SUIF

This section describes DEP_AN, a dependence analysis pass that generates dependence information which can be used by other optimization passes. DEP_AN itself does not optimize any code; it only adds annotations to the code which can be read by later optimization passes that require the dependence information.

4.3.1 Motivation

The development of DEP_AN was principally motivated by the dependence information requirement of the new post-optimizer for its software pipelining optimization. Since software pipelining involves reordering or rescheduling instructions in a loop, it needs to know any dependency that may exist between a pair of instructions within an iteration or across the iterations before reordering them.

There is another optimization pass, besides the post-optimizer, that uses the dependence information generated by DEP_AN. This is the common array sub-expression elimination pass which is added to INDVAR and is described later in Section 4.4.

The dependence analyzer also does a restricted form of alias analysis by attaching “alias” annotations to instructions that use address values. An annotation indicates the scalar variable or array variable that is referenced by the corresponding address value. The alias information is used by both the post-optimizer and the DSP simulator to obtain the variable names of the address operands used in the instructions.

In short, the development of DEP_AN was motivated by the need for dependence information by the two optimization passes described above and the need for alias information by the post-optimizer and the DSP simulator.

4.3.2 DEP_AN: Some Highlights

This section briefly explains the concept of dependence in the context of array accesses. Then it briefly describes how DEP_AN works.

It is assumed that the reader knows about true, anti and output dependencies between instructions or accesses. Refer to [BGS93][ASU88] which cover this topic in more detail. As an example, consider the following code fragment:

In this piece of code, `expr1` and `expr2` are assumed to be expressions involving the loop index `i`. There are two accesses of the array `a` in an iteration of the `for` loop: *access 1* is a write access to `a`, and *access 2* is a read access to `a`. There are a number of dependencies that may exist in this kind of situation. Within an iteration, *access 2* may depend on *access 1* if `expr1` and `expr2` evaluate to the same value. There could also be a dependence of *access 1* on itself across iterations if `expr1` evaluates to the same value in two different iterations.

```
for (i = 0; i < N; i++) {
  a[expr1] = expr3;    /* access 1 */
  x = a[expr2];        /* access 2 */
}
```

Figure 4.9 Dependence between array accesses - an example

Similarly there could be a dependence either way between *access 2* and *access 1* across iterations if **`expr1`** and **`expr2`** evaluate to the same value in different iterations. To find out if an array access *x* depends on an array access *y* in general, the dependence analyzer must examine the locations of the array accesses, the contexts of these accesses, including all the enclosing loops, the index expressions of these array accesses, and the relationship of the index expressions to the loop index variable.

We will now see how DEP_AN determines dependence between array accesses. Two things to note here are that DEP_AN is a modified form of a sample program found in the SUIF "cookbook" [SUC94] and secondly DEP_AN calls a function provided by the SUIF library. This function, called **`DependenceTest`**, accepts two array accesses as arguments and returns the dependence information to the caller. It is this function that does all the analysis required to compute the dependence information. This information consists of a flag indicating the availability of the dependence information and a set of distance vectors, each vector showing the distance from the first array access to the second in the iteration space corresponding to all the enclosing loops. Refer to the documentation on the SUIF Dependence library [SUC94] for more information on the function **`DependenceTest`**. All that DEP_AN has to do is collect all the array accesses inside all the outermost **`for`** loops of a procedure and use **`DependenceTest`** to find the dependency between any two of the array accesses. The dependence information thus obtained is attached as annotations to the access that is the source of the dependency. For example, if access *y* is dependent upon access *x*, meaning access *x* occurs earlier in time than access *y*, then the annotations are attached to access *x*.

The first part of DEP_AN consists of collecting all array accesses in a procedure by using two functions with the same name **`find_array_instr`**. The first one traverses a tree of syntax nodes of type **`tree_node`** and the second one traverses an expression tree, which consists of **`instruction`** nodes, inside a syntax node to identify array instructions. Figure 4.10 shows the pseudo code of the first **`find_array_instr`** function. It recursively calls itself to traverse the tree of syntax nodes until it reaches a leaf **`tree_instr`** node, when it calls the second **`find_array_instr`** function. Note that when it has finished processing an outermost **`for`** loop, it calls the function **`process_array_instr_list`** which processes the list of array accesses. Figure 4.10 also shows the pseudo code for the second **`find_array_instr`** function. It recursively calls itself to traverse the expression tree starting from the root, and whenever it finds an ***array*** instruction, it adds it to **`ai_list`**. And whenever an instruction uses the address of a scalar or an array, it attaches the alias annotations containing the name of that scalar or array to the instruction.

Figure 4.11 shows the algorithm of function **`process_array_instr_list`** in the form of a flow-chart. This function processes the array instructions that were collected in a list by the **`find_array_instr`** functions as described earlier. This function uses two nested loops in such a way that it obtains all possible pairs of instructions and it calls the function **`DependenceTest`** for each pair to determine the dependency for that pair. If there is a dependency, it attaches a "dependency" annotation to the source of the dependency which contains the dependence information.

Note that, because DEP_AN performs its analysis on array accesses, the DEP_AN pass has to be run before arrays are dismantled for the purpose of running the INDVAR optimization pass. In any case DEP_AN has to be run before INDVAR, because INDVAR contains another optimization that we call 'common array sub-expression elimination' that requires the dependence information generated by DEP_AN. In the next section we describe the common array sub-expression elimination optimization.

4.4 Common array sub-expression elimination

This section describes the common array sub-expression elimination (abbreviated as ACSE) optimization, its motivation, and its implementation. Note that this optimization has been added to the INDV AR phase, because ACSE uses many of the same data structures and logic that the induction variable creation optimization uses.

4.4.1 What is ACSE?

The common array sub-expression elimination optimization is an optimization that seeks to eliminate redundant memory loads within an iteration of a **for** loop. As an example, consider the code fragment shown in Figure 4.12. In this piece of code, the array element **a[i]** is read twice within a single iteration. The second read access to **a[i]** is clearly redundant because the value of **a[i]** in the second read access is the same as the value read in the first access. We can safely say that the second read of **a[i]** is redundant because:

```
find_array_instr(treeNodeList, ai_list)
{
   for (every tree_node in treeNodeList) do {
     if (current tree_node is tree_instr) {
        /* contains expression tree or instructions */
        instr = root instruction in current tree_node;
        call find_array_instr(instr, ai_list);
     }
     else {
        /* means non-leaf node - has other tree_node_list */
        call find_array_instr(child treeNodeList, ai_list);
        /* for every child tree_node_list present */
        if (current tree_node is an outermost for loop) {
           call process_array_instr_list(ai_list);
        }
     }
   }
}


find_array_instr(instr, ai_list)
{
   if (instr is an array instruction) {
     add instr to ai_list;
   }
   if (instr requires alias annotation) {
     vname = array or scalar name used in instr;
     attach alias annotation containing vname to instr;
   }
   for (each source operand of instr) do {
     if (current source operand is an instruction) {
        sinstr = instruction in the current source operand;
        call find_array_instr(sinstr, ai_list);
     }
   }
}
```

Figure 4.10 The two functions `find_array_instruction`

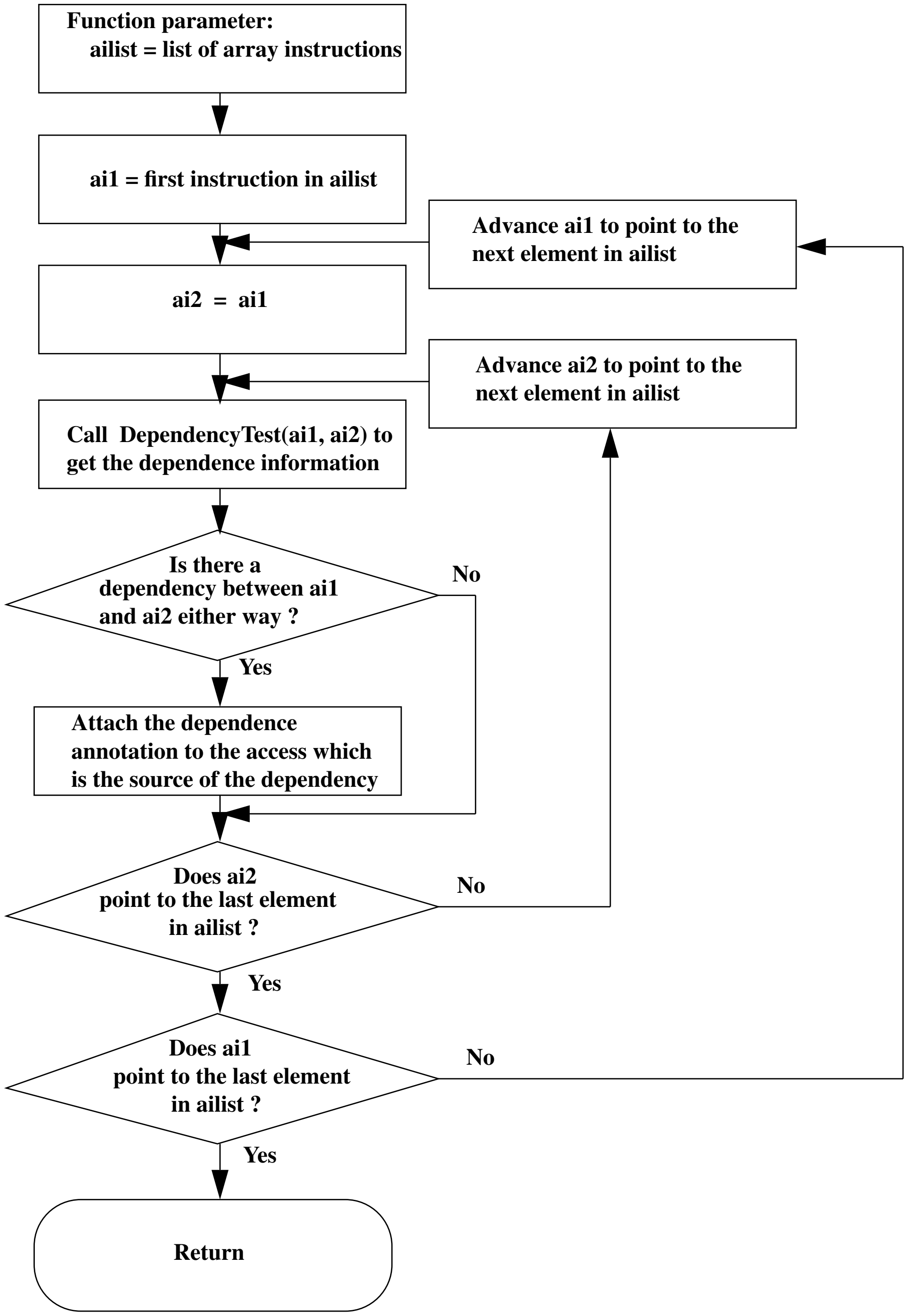

Figure 4.11 The function `process_array_instr_list`

```
for (i = 0; i < N; i++) {
   x = a[i] + m * x;   /* access 1 */
   y = x * a[i];       /* access 2 */
}
```

Figure 4.12 An example of a redundant array access.

```
for (i = 0; i < N; i++) {
           __tmp1 = *__ivar1;    /* copy to a temp */
           x = __tmp1 + m * x;   /* access 1 */
           y = x * __tmp1;       /* access 2 */
        }
```

Figure 4.13 After removing the redundant array access

- the first read of `a[i]` always occurs when the second read occurs, and
- there is no store to `a[i]` between the two read accesses of `a[i]`.

On the other hand, had the first access been inside an `if` block, its occurrence would have been conditional. Also if there was a store to some element `a[expr]` between the two read accesses, such that `expr` could equal `i`, then we cannot safely say that the second read of `a[i]` reads the same value as the first one. So in either of these cases, the second read would not be redundant.

But in the above case, since there was a redundancy, we can eliminate the second read access by copying the value from the first read access. The above piece of code would look as shown in Figure 4.13 after performing ACSE assuming that the induction variable creation optimization has already converted the array expression `a[i]` into an induction variable `__ivar1`.

4.4.2 Motivation

We decided to implement the ACSE optimization because it appeared to be a simple enough optimization to implement as an addition to the INDVAR pass considering the availability of the dependence information from the DEP_AN pass. The major benefit of this optimization to the DSP application programmers is that they can write their code in the more natural fashion of using and reusing array accesses without having to worry about its effect on performance. In the absence of this optimization, the application programmers would typically be doing this optimization manually in their programs by using temporary variables to eliminate redundant array accesses.

4.4.3 Implementation

As mentioned earlier, this optimization has been added to INDVAR, the induction variable creation optimization. In the following description, the part of INDVAR that deals with the ACSE optimization is referred to as the ACSE logic.

When INDVAR traverses an expression tree to search for maximal linear or loop-constant expressions in order to replace them with induction variables, the ACSE logic looks at the parent or destination instruction of an linear expression to determine if it is a redundant ***load*** instruction. It is able to determine this because INDVAR already maintains a list of available linear expressions for the current loop. This list has a few additional fields of information which the ACSE logic can use to determine that a read access on an array element has occurred earlier in the loop, and the read value can be reused because there was no store to that array element until the current point. For example, the ACSE logic uses the

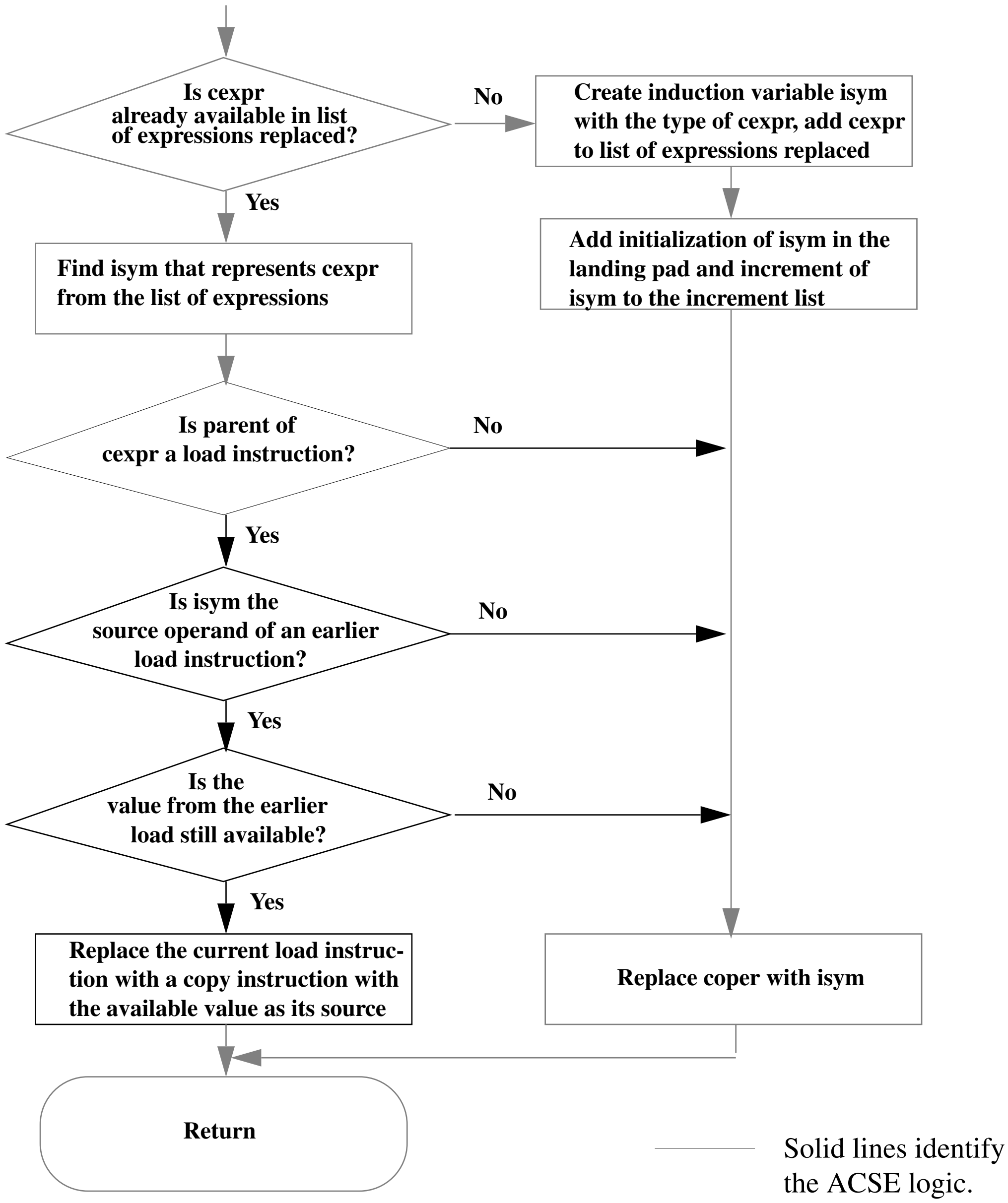

Figure 4.14 The ACSE logic added to `replace_operand`

"dependence" annotations added by the DEP_AN phase to determine if a given ***store*** instruction can write to the same array element that was read earlier, thereby making the value read from that array element unavailable. In case of a redundant ***load*** instruction, it creates a temporary variable, unless already created, to copy the value of the earlier ***load*** instruction and replaces the current ***load*** with a ***copy*** instruction with the temporary variable as its source. Later on, the copy propagation pass removes any redundant ***copy*** instructions.

Figure 4.14 shows the flow-chart of the part of the function **`replace_operand`** that was modified to include the ACSE logic. The diamonds and a box that are shown with solid borders and the solid arrow-lines constitute the part that differs from Figure 4.8. This is the place where **`replace_operand`** finds that a linear expression is available in the list of linear expressions that have already been replaced in the current loop. In the modified

```
reset_available_expr(expr_list, instr_store)
{
   for (every expression in expr_list) do {
      if (expression was used in load instruction) {
         if (instr_store annotations show dependence on
                 expression) {
            mark load value from expression as not available;
         }
      }
    }
}
```

Figure 4.15 Pseudo-code of the function `reset_available_expr`

function, it then proceeds to find out if the parent instruction of the current linear expression is a redundant ***load*** instruction that it can eliminate.

The ACSE logic also includes a function **`reset_available_expr`** which is called by the function **`process_instruction`** when it is processing a ***store*** instruction. The pseudo-code of **`reset_available_expr`** is shown in Figure 4.15. It goes through the list of linear expressions and for every expression that is used in a ***load*** instruction, it looks at the annotations of the ***store*** instruction. If the annotations indicate a dependency on the ***load*** instruction, such that the ***store*** instruction could modify the location pointed by the ***load*** instruction, it marks the value of the ***load*** instruction as not available.

4.5 Summary

In this chapter we described the implementation of two optimization passes, namely , induction variable creation and common array sub-expression elimination plus an analysis pass called dependence analysis. Figure 4.16 shows the location of these passes in the retargeted SUIF compiler , which is invoked using the command **`scc-DSP`**.

The dependence analysis pass attaches annotations to the SUIF code, which are present in the assembly output generated by DGEN in the **`.s`** file. However the old post-optimizer did not have the capability to read these annotations. In the next chapter we see how this capability was added to the post-optimizer by developing a new front-end.

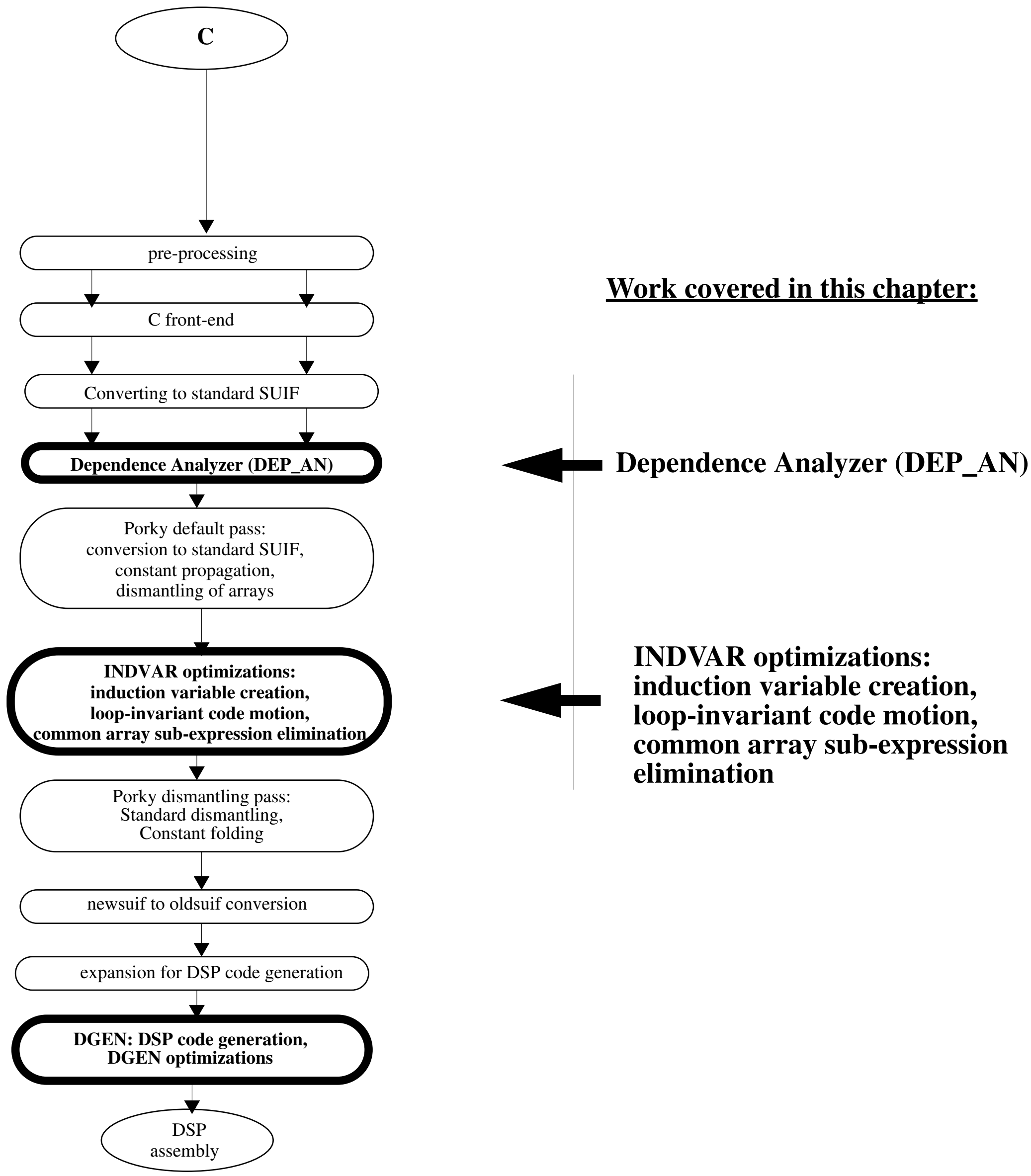

Figure 4.16 Structure and phases of the retargeted SUIF Compiler

Chapter 5 Post-optimizer Front-end

This chapter describes a new *lex*- and *yacc*-based front-end that was developed for the post-optimizer. We present the motivations behind this development and the implementation of the front-end.

5.1 Motivations

As described in Chapter 2, the post-optimizer reads the assembly output file generated by either the GCC-based or the SUIF-based compiler and performs some DSP-specific optimizations on it to generate optimized and compacted code which the DSP simulator can read and execute. Since the input to the post-optimizer is an assembly language text file, the front-end of the post-optimizer is essentially like a parser of an assembler. However the old front-end of the post-optimizer parsed its input using normal C library I/O functions and a series of **`strcmp`** function calls. This technique, besides being primitive, has a number of drawbacks. Firstly, this parser is very sensitive to white-spaces in the input stream and hence is very restrictive with respect to the format of the input code that it will parse. For example, adding a white-space between two operands of an instruction will prevent the post-optimizer from processing the assembly file. Secondly, changing the grammar of this parser is difficult because the grammar is hard-coded in the **`strcmp`** function calls and the surrounding C code, making the grammar quite unstructured. Also the unstructured nature makes maintaining such a parser difficult as compared to a parser that is generated from a parser generator such as *yacc*. Finally, this kind of parser cannot generate good diagnostics and cannot recover from syntax errors which a good compiler or assembler should be able to do.

It is because of these drawbacks that we decided to develop a new parser for the post-optimizer using a parser generator. And, of course, the need to have a new parser for the post-optimizer became paramount when there was a requirement for the post-optimizer to be able to read the annotations in the code generated by the new SUIF compiler. The post-optimizer needs these annotations for implementing software pipelining.

It was natural for us to choose *lex* and *yacc* to develop the new parser because of their advantages. First of all, these tools are available on any Unix system. Secondly, *yacc* is a powerful parser generator which can generate a parser for any grammar that is LALR(1). See [ASU88] for a detailed description of context-free grammars and LALR(1) grammar. Thirdly, the user-supplied semantic routines in a *yacc*-generated parser allow better diagnostics and syntax error recovery. Also *yacc* supports a non-terminal called **`error`** which allows the parser to recover from syntax errors. Finally the input files to *lex* and *yacc* serve as the definition of the grammar of the language that we are parsing. An added advantage of lex and yacc is that the debug modes of both *lex* and *yacc* enable easy debugging of the parser and the grammar.

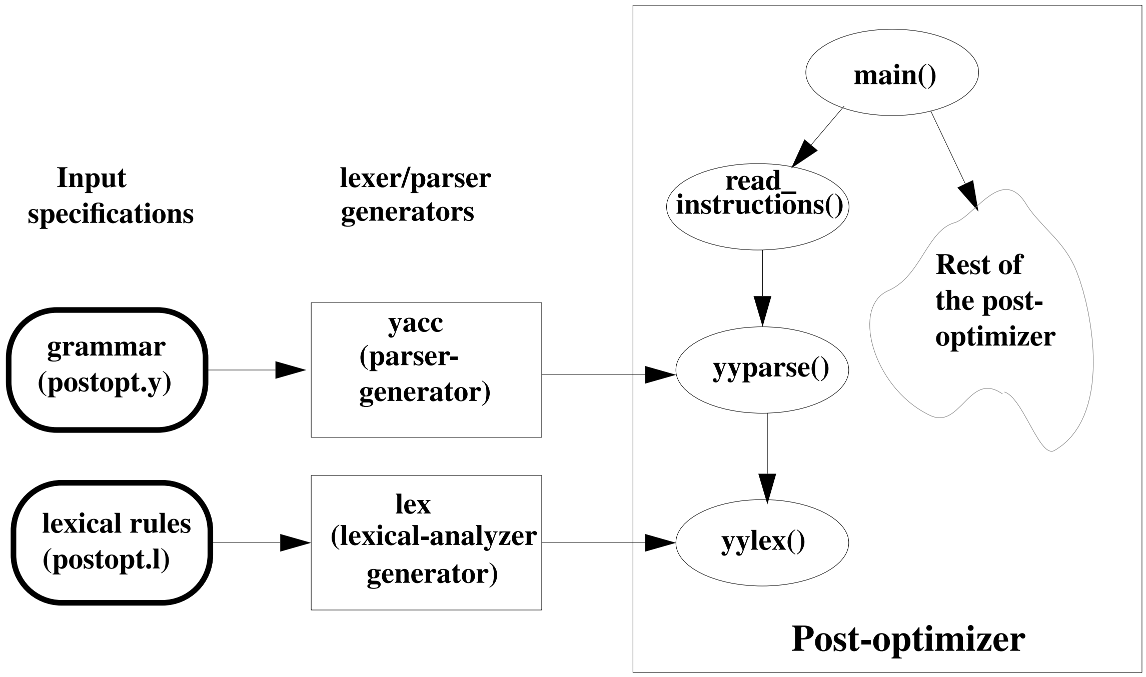

Figure 5.1 The post-optimizer with the new front-end

5.2 Implementation

Figure 5.1 depicts the manner in which ***lex*** and ***yacc*** are used. The ovals with thick borders indicate the input specifications we developed for the new parser. The big box on the right hand side represents the post-optimizer which uses the parser and the lexical analyzer generated by ***yacc*** and ***lex*** respectively.

The process of developing a new front-end or a parser using parser generators involves the following steps [Bis91]:

- Formally specify a grammar in a form recognized by the parser generator. For each grammatical rule in the language that you are parsing, describe an action that is to be taken when an instance of that rule is recognized. The grammar for our assembly language is specified in the file **`postopt.y`**.

- Develop a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer can be produced using tools such as ***lex*** or ***flex***. The rules for our lexical analyzer are specified in the file **`postopt.l`**.

- Add the controlling function that calls the parser. We changed an existing function **`read_instructions()`** in the file **`instructions.c`** to call the parser function **`yyparse()`**.

- Add error reporting routines that are needed by the generated parser. In our case, these have been included in the file **`postopt.y`**.

- Run the parser generator, ***yacc***, on the grammar specification to produce your parser.

- Compile and link the parser generated by the parser generator into the post-optimizer.

The next sub-section describes the first step in the development of the new front-end, which is the specification of the grammar for the assembly language that our front-end is going to parse.

5.2.1 Specification of the Grammar for *yacc*

yacc, which stands for 'yet another compiler compiler', is a parser generator that reads a grammar specification and if that grammar happens to be a context-free LALR(1) grammar, it generates an LALR(1) parser from it. The input grammar specification to *yacc* is simply a set of rules specified in a machine readable Backus-Naur Form. A rule typically is a "syntactic grouping" and a specification of how it is constructed from its parts[Bis91]. The specification also includes a "semantic action" which specifies the processing that needs to be done when the parser applies that rule during parsing[ASU88]. For the remainder of this chapter the reader is assumed to be familiar with parsers, parser generators and related terminology.

The process of developing the grammar specification for the post-optimizer front-end was relatively straight-forward because of the fact that assembly language grammars are simpler than those for high-level languages such as C. Assembly languages do not have complex constructs which are present in high-level languages. Appendix B shows the grammar that we developed for the DSP assembly language that is the language of the programs processed by the post-optimizer. The remainder of this section describes the main elements of this grammar and some of the main design decisions that we made during the development of this grammar. Note that, in the following discussion, a grammar entity is shown by a symbol in bold font, where the symbol is the name of the corresponding non-terminal or terminal used in the grammar. Following the *yacc* conventions, non-terminals are denoted in lower-case and terminals or tokens in upper-case.

A DSP assembly program (**`program`**) consists of external definitions (**`extern_def`**). An **`extern_def`** is one of function (**`function`**), data definition (**`data_def`**) or assembler directive (**`asmdir`**). A **`function`** consists of a begin directive (**`asmdir_begin`**) followed by one or more statements (**`statement`**) and an end directive (**`asmdir_end`**). The semantic action for the production for a **`function`** checks that the function names in both the begin directive as well as the end directive are the same, otherwise it reports it as a warning. A **`statement`** is one of instruction (**`instruction`**), assembler directive (**`asmdir`**) or label statement (**`LABELTYPE`** and **`COLON`**). The production for **`instruction`** shows all the possible instruction formats ranging from **`zero_op_instruction`** to **`four_op_instruction`**. A typical instruction format consists of an **`OP_CODE`** and zero or more operands; and an operand is one of a register (**`REG`**), an integer (**`INTEGER`**) or a symbolic address (**`LABEL`**). The semantic action corresponding to the production for each of the instruction formats has calls to the functions **`match_opcode_format`** and **`check_reg_type`** to check that the actual value of the **`OP_CODE`** is compatible with the instruction format and the types of the register operands supplied in that instruction. For example, the op-code **`ld.s`** can occur only in the instruction format identified by the non-terminal **`two_op_first_indirect_second_reg_instruction`** and for that op-code the first register has to be an address register and the second one has to be a float register.

One of the main reasons for developing the new front-end was the requirement of parsing and incorporating the SUIF annotations present in the input assembly file. In the internal SUIF representation, annotations are 'attached' to various SUIF objects such as instructions, syntax nodes, procedures or variable declarations. In the assembly file generated by DGEN, these annotations immediately follow the assembly language representation of the SUIF objects to which they were attached. For example, if a SUIF instruction has annotations attached to it, DGEN transfers these annotations to the translated DSP instruction, and prints out the annotations immediately after the DSP instruction has been printed out. Each annotation appears on a separate line and starts with the character **`'#'`**. This annotation line is described by the following production in the grammar:

`annotation_line: '#' '[' ANNOTE_STRING zero_or_more_annotes ']' NL`

where **`ANNOTE_STRING`** is a string token identifying the name of the annotation, and **`zero_or_more_annotes`** is a non-terminal which indicates the presence of zero or more *icons*. An *icon* is a textual representation of any type of a SUIF object. A sample annotation line is shown below:

`#          ["instr_number" 21]`

This is an "instr_number" annotation showing the instruction number assigned by SUIF to the instruction which precedes this annotation.

The non-terminal **`annotations`** stands for zero or more **`annotation_line`**s. Because annotations can be attached to various SUIF objects, the non-terminal **`annotations`** is present in the productions of many non-terminals such as **`program`** and **`data_def`**.

5.2.2 Development of the Lexical Analyzer

A parser generated by ***yacc*** makes calls to a function **`yylex`** to get tokens from the input stream. The function **`yylex`** can either be hand-written or generated from a lexical analyzer generator such as ***lex***. In this work, we decided to use a lexical analyzer generator called ***flex*** which is upward compatible with ***lex***. The main advantage of ***flex*** over ***lex*** is that ***flex*** supports a feature called *exclusive start condition* which is not supported by ***lex***. This feature is very useful in creating exclusive lexical analyzer states for reading comments or annotations. The *exclusive start condition* is a way of making the lexical analyzer aware that it is reading special patterns such as comments or annotations wherein it should suppress the application of normal pattern matching rules.

Appendix C shows the input specification to ***flex*** that generated our lexical analyzer. This specification consists of several pattern matching rules and their associated actions to identify tokens in the input stream. A pattern matching rule is somewhat similar to the pattern matching specification used in the ***grep*** command to search for strings in a text file. The action associated with a rule typically extracts the relevant information from the text that matched the pattern specified and returns the associated token to its caller, which ultimately is the parser. For example, the rule that identifies a register (**`REG`**) token in the input stream appears as:

`[adf][0-9]{1,2}`

This rule identifies a string pattern which starts with one of **`'a'`**, **`'d'`** or **`'f'`** and has exactly one or two decimal digits following that character. The action for this rule, looks at the character and the decimal number following it to convert that into a numerical value understood by the parser. For example, **`d5`** becomes 5, **`a4`** becomes 36 and **`f2`** becomes 66.

As mentioned earlier, this specification uses *exclusive start conditions* to read comments, annotations, data definitions and assembler directives. For example, when the lexical analyzer scans the annotation start character **`'#'`**, the action for that rule sets the *exclusive start condition* **`<annote>`**. In this state the lexical analyzer will only apply those rules which have been prefixed with **`<annote>`** which means that such rules can be applied only in the start condition **`<annote>`**. In this way the lexical analyzer is able to recognize ordinary strings as annotation icons rather than something else.

The rest of the input specification is quite straight-forward and has rules to identify various tokens used in the grammar described in the previous sub-section.

5.2.3 Interface with the Post-optimizer

There were only two main aspects to consider while interfacing the new front-end with the existing post-optimizer. The semantic actions corresponding to various productions in the grammar were responsible for constructing instruction or data definition objects as they are represented in the post-optimizer. As an example, the semantic action for the production for the non-terminal **`two_op_both_reg_instruction`** constructs the instruction object of type **`INSTRUCTION`** from the semantic values of the terminals and non-terminals of the production. It also creates an object of type **`OBJECT`** which is the basic unit for representing instructions and data definitions in the post-optimizer. An object of type **`OBJECT`** contains a pointer to the instruction object when the object is supposed to be representing an instruction. The types **`INSTRUCTION`** and **`OBJECT`** are defined in the post-optimizer header files which are also included in the parser. The parser builds the linked list of **`OBJECT`**s as it parses the input program and after reaching the **`goal`** symbol (which indicates successful completion of the parse), it initializes the **`FIRST_OBJECT`** and **`LAST_OBJECT`** variables to point to the first and last objects in the linked list respectively.

The only other aspect of the interface is the **`yyparse`** function that the post-optimizer needs to call in order to parse the input file. The post-optimizer has a function called **`read_instructions`** to read and parse the input file. The code in **`read_instructions`** is now replaced with a call to **`yyparse`**. When **`yyparse`** returns, the post-optimizer has the parsed program available to it as a linked list of **`OBJECT`**s through the variables **`FIRST_OBJECT`** and **`LAST_OBJECT`**.

5.3 Summary

To make the information in the SUIF annotations available to the post-optimizer, we had to develop a new front-end for the post-optimizer that could read and parse the SUIF annotations. In this chapter we covered the development of this new front-end.

Previous chapters described how the SUIF-based compiler was retargeted to the DSP architecture. W ith the post-optimizer fitted with a new front-end, it became possible for us to use the DSP development/ simulation system to compile and run the various DSP benchmarks with both the GCC-based and SUIF compilers and compare the performance of the code generated by each of them. The next chapter describes the results obtained from this performance analysis.

Chapter 6 Performance

Previous chapters described the process of retargeting the SUIF compiler to the DSP architecture and how the former was incorporated into the DSP development and simulation system. This chapter summarizes the results of the performance analysis we made of the code generated by the new retargeted SUIF compiler vis-a-vis the code generated by the existing GCC-based compiler.

We first describe the experimental framework we set up to conduct benchmark testing and to gather performance results. Next we present the performance results obtained from compiling and running the benchmarks using both compilers under various conditions. The results are presented through a series of graphs which give a better idea of the comparative performance rather than a table of numbers because of the visual presentation. The presentation of results is accompanied by a discussion of various optimization passes, their impact and performance problems with the SUIF-generated code. The chapter ends with some conclusions drawn from the analysis we did.

6.1 Experimental Framework

Before presenting the results obtained from our performance analysis, it is necessary to describe the framework we used for conducting our performance testing. This section describes the methodology and the set-up we used to conduct our performance testing and discusses our choice of graphs as a way to present the performance results we obtained.

6.1.1 Methodology

We used a fixed set of eighteen benchmarks divided into twelve DSP kernels and six DSP applications. The original benchmarks were obtained from previous work done on the DSP architecture [Sin92][Sag93]. As mentioned earlier in Chapter 2 and Chapter 4, these benchmarks were written in the 'DSP-style' to get better performance out of the GCC-based compiler. We call this original version of the benchmark *DSP-style with SWP* where SWP is an abbreviation for software pipelining, which was described in Section 2.3. We created two more versions of these benchmarks: the first version is called *C-style* and the second one is called *C-style with SWP*. Both the *C-style* and *C-style with SWP* versions are semantically equivalent to the *DSP-style with SWP* version. In our performance testing, we ensured that, given the same input, these new versions of the benchmarks produced the same output as the *DSP-style with SWP* version. The *C-style* benchmarks do not have the constraints of the DSP-style and look more like natural C language code, whereas the *C-style with SWP* version, as the name suggests, is the software pipelined version of the *C-style* benchmarks.

An interesting aspect of our experimental framework is the presence of three versions of the benchmarks. A cursory survey of literature on the topic of compiler optimizations revealed that this method of creating different versions of benchmarks to test the optimizing capability of a compiler is rare. What researchers and investigators often do is to test their optimizer on a fixed set of benchmarks and compare the performance of the generated code for those benchmarks with and without the optimization [TjH92][EGK94][GrM94][Mcf91][PeH90]. It is to be noted, however, that our goal is somewhat different from the goal of all these researchers. We are implementing an optimization in a new compiler that allows us to rewrite existing benchmarks in a more readable, or what can be considered as natural C, style without sacrificing performance. To find out how well our new optimizing compiler was working, we needed to create additional versions of our benchmarks. The *DSP-style with SWP* version contains many hand-coded optimizations, such as induction variable creation and software pipelining. These hand-coded optimizations were made necessary in the earlier work [Sin92][Sag93] because of the fact that this work explored some of the novel optimizations such as hardware do-loop generation that were closely tied to the DSP architecture. This optimization, which takes place in the post-optimizer, is dependent upon the front-end compiler being able to transform the **`for`** loops to eliminate references to the loop index variable. Because the GCC-based compiler that was used in this work did not perform this optimization, these benchmarks had to be written in a specific way that came to be known as the 'DSP-style'. It is our objective to show that an optimizing front-end compiler can in fact optimize DSP benchmarks written in a natural C fashion to give most, if not all, of the performance benefits that we obtain from writing the benchmarks in the 'DSP-style'. In short, with our new optimizing compiler we are attempting to improve the readability of the DSP applications without sacrificing performance. This required us to create the *C-style* version of the benchmarks to undo the hand-coded optimizations of the *DSP-style with SWP* version. Because we did not have the software pipelining optimization in either the front-end or the back-end compiler, we also created another version called *C-style with SWP* from the *C-style* version of the benchmarks to compensate for the lack of software pipelining in the *C-style* benchmarks.

6.1.2 Set-up

All three versions of a benchmark – *DSP-style with SWP*, *C-style* and *C-style with SWP* – reside in a single directory assigned for that benchmark. The name of the *C-style* version of a benchmark uses prefix **`S`**, whereas that of the *C-style with SWP* version uses prefix **`P`**. For example, if the *DSP-style with SWP* version of a benchmark is named **`iir_4_64.c`**, then its *C-style* version would be named **`Siir_4_64.c`** and the *C-style with SWP* version would be named **`Piir_4_64.c`**. All the kernel benchmark directories are under the directory **`kernel/`** and all the application benchmark directories are under the directory **`applications/`** and both of these are under the directory **`test_suite/`**. Hence all the benchmarks and all of their versions are stored in a directory hierarchy that is under a directory called **`test_suite/`**.

We developed a Perl [Per91] script called **`perf_dsp.pl`** that traverses the directory hierarchy under **`test_suite/`**, finds directories that have valid benchmarks in them, compiles and runs all versions of the benchmarks using both the compilers, collects the performance data generated by the DSP simulator ***dspsim*** and generates a report and other output files. These other output files that **`perf_dsp.pl`** generates can be directly used as input to the plotting program ***gnuplot*** to generate the plots that are presented in this chapter.

6.1.3 Presentation of Results

When we made a brief survey of some recent literature that discussed compiler optimizations, we noticed many different ways in which the performance results were presented. The most common was a tabular presentation of performance numbers such as instruction cycle count or elapsed time in seconds for various benchmarks[EGK94][GrM94][TjH92]. These tables would usually contain columns that show the speedup or savings in the performance due to the use of a particular optimization[GrM94][TjH92]. On the other hand some of the literature only presented the speedup numbers or percentage performance improvement as a result of using their optimization[GrM94][PeH90].

There were a few papers that chose to present their performance results through graphs

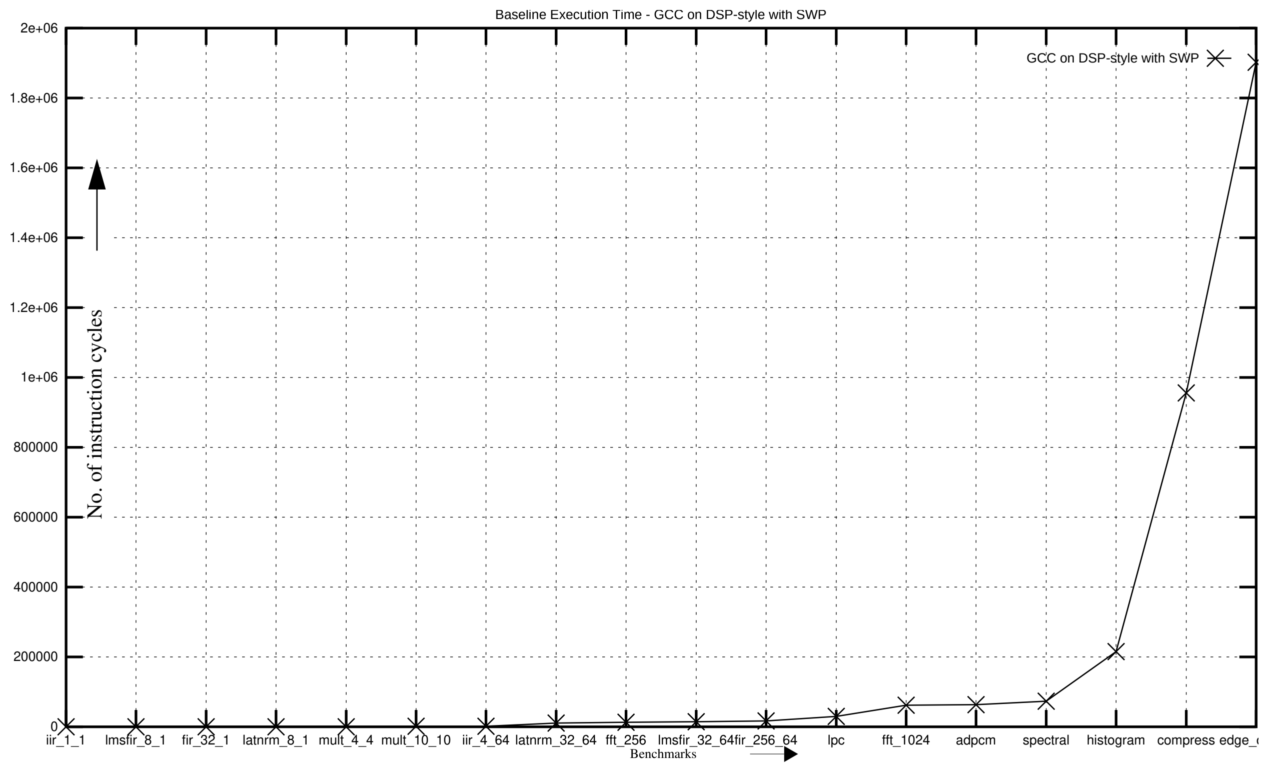

Figure 6.1 Baseline execution time - GCC on *DSP-style with SWP* benchmarks

[WoJ92] or bar charts[Mcf91]. Graphs were used to plot the performance improvement, on the y-axis, against some numerical parameter such as number of processors on the x-axis[W oJ92]. The way we have chosen to present our performance results, comes closest to [Mcf91], although we have used a line graph instead of the bar chart of [Mcf91]. We find that a line graph gives the best visual impression of a performance comparison, when we try to include more than two sets of performance data for the sake of comparison. A bar chart, on the other hand, tends to clutter up the diagram and is not as compact as a line graph when multiple data-sets are used.

6.1.4 Baseline Execution Time

Figure 6.1 shows a graph that plots the instruction cycle count, as reported by ***dspsim***, for each of the eighteen *DSP-style with SWP* version benchmarks compiled with the GCC-based compiler. The benchmark names are shown on the X-axis and are arranged in the ascending order of the instruction cycle count for the benchmark. The Y-axis shows the corresponding instruction cycle count for each of them. All eighteen points are joined with a line, and because the benchmarks are in the ascending order of their instruction cycle count on the X-axis, the line rises from left to right and has positive slope at each point. We call this the baseline graph because we are going to compare all of our execution time numbers, which are obtained under different conditions of compilers, options and versions of benchmarks, with the numbers that are shown in this graph. In fact all the later graphs, except for the one in Figure 6.2, plot the relative execution time of a benchmark as opposed to the absolute instruction cycle count for that benchmark. The relative execution time is defined as the instruction cycle count for the benchmark divided by the instruction cycle count for the *DSP-style with SWP* version of that benchmark compiled with the GCC-based compiler. For example, if **`iir_4_64.c`** takes 1683 cycles to run when compiled with the GCC-based compiler, our baseline number for **`iir_4_64.c`**, and if **`Siir_4_64.c`** takes 10343 cycles to run when compiled with the SUIF compiler without any optimization, then the relative execution time of the *C-style* version of the benchmark under the above conditions is 10343 ÷1683=6.1455734. Because the instruction cycle count spans a wide

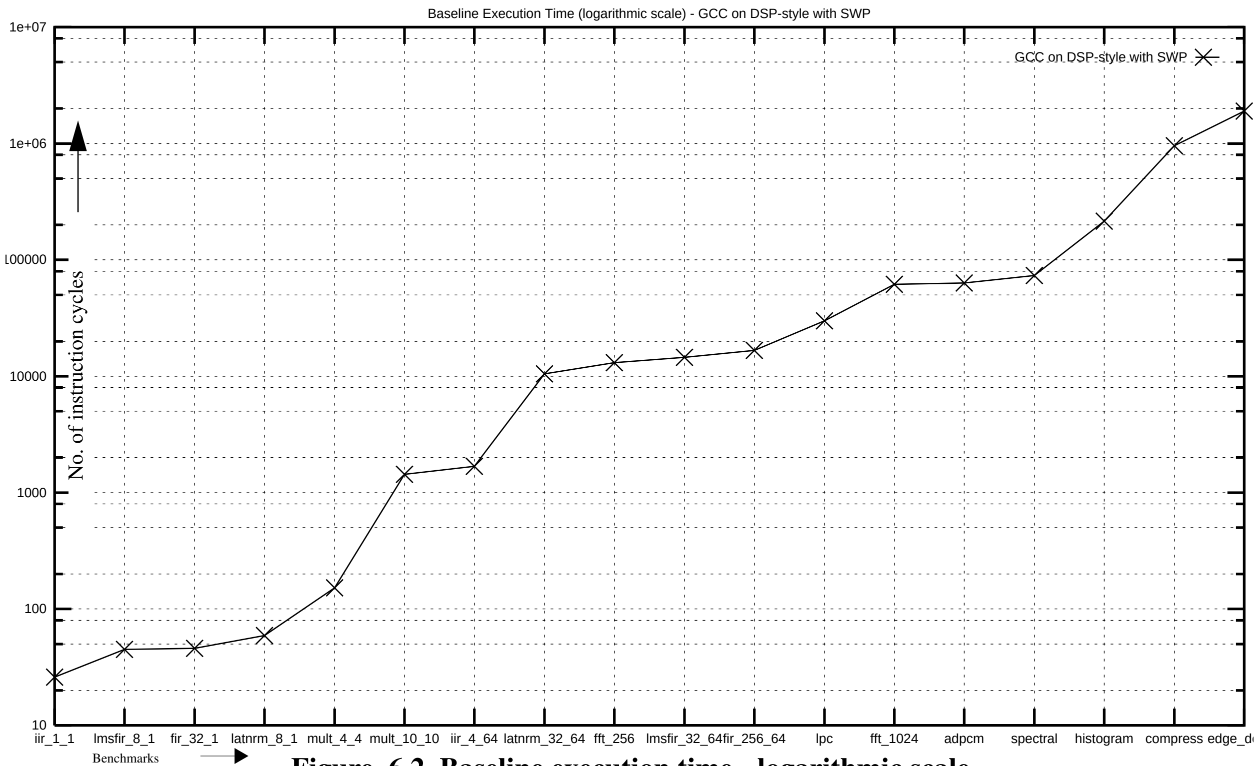

Figure 6.2 Baseline execution time - logarithmic scale

range from 22 cycles up to around 2 million cycles, the graph in Figure 6.1 appears compressed and flat on its left half. The same graph is shown in Figure 6.2 with the Y-axis representing instruction cycle count on a logarithmic scale. This gives us an indication of the variation in the instruction cycle count on the lower end of the graph.

6.2 Results and Analysis

This section presents the performance results that we obtained from running all three versions of the benchmarks using both the compilers for various combinations that consisted of choice of benchmark version, choice of compiler and choice of optimization(s) for the SUIF compiler . As explained in the previous section we have shown the relative execution time with respect to the baseline execution time for all of our results. The order in which we have presented these results roughly parallels the stages in our development process. We start with the results we obtained from the SUIF compiler when DGEN first started generating code for the DSP architecture successfully. From then on, at each stage we analyzed the SUIF-generated code to find inefficiencies and tried various optimizations to improve the performance. This section also discusses our decision making and efforts with regard to adding various optimizations to the SUIF compiler based on our analysis of the SUIF-generated code.

6.2.1 GCC Compiler on the *C-style* benchmarks

Before we analyze the performance of the SUIF code, let us look at the performance of the code generated by the GCC-based compiler for the *C-style* benchmarks. Figure 6.3 shows a graph of the relative execution time of this code. We can see that the relative execution time varies from the best of almost 1.0 for `iir_1_1`, to the worst of almost 8.0 for `fir_256_64`. Note that the *C-style* benchmarks were compiled with the GCC-based compiler version 1.39 and the GCC optimization option `-O` turned on. We were unable to get

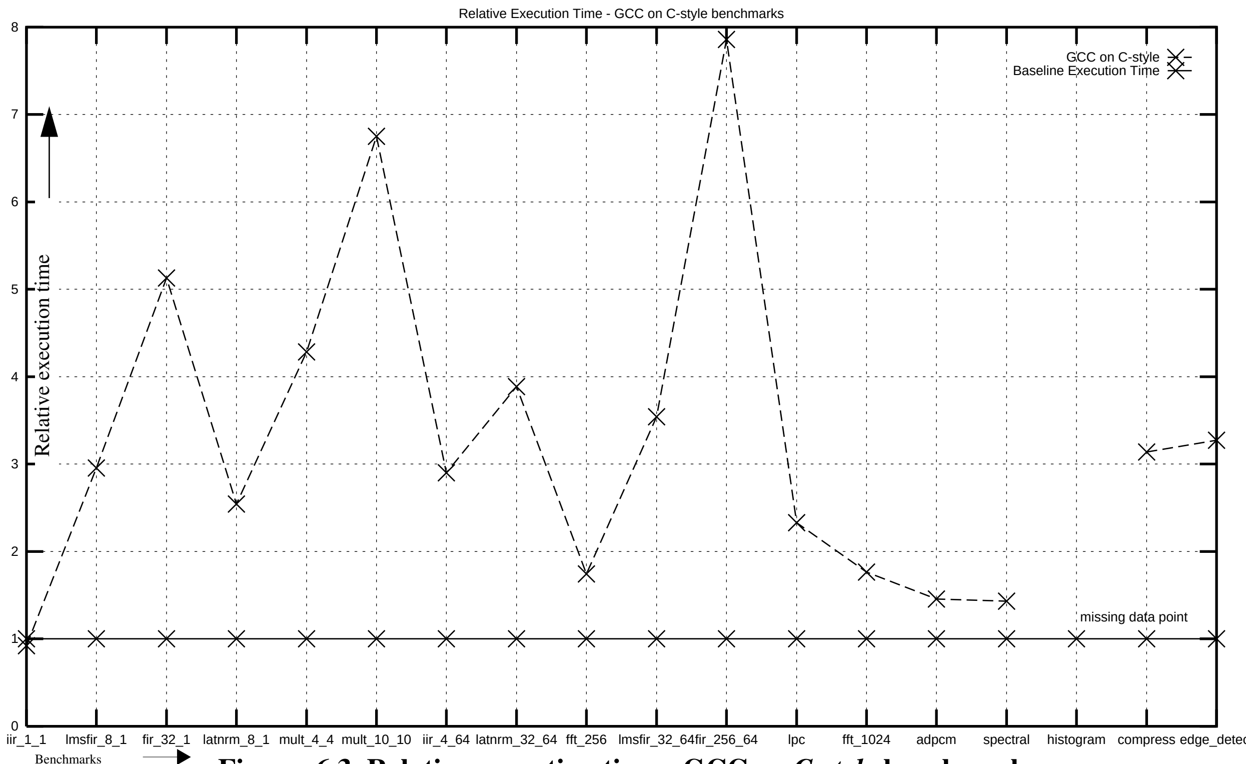

Figure 6.3 Relative execution time - GCC on *C-style* benchmarks

this GCC-based compiler to compile with higher optimization options such as **`-O2`** and **`-O3`**. Also note that there is a missing data point, corresponding to the benchmark **`histogram`**, in the graph because the GCC-based compiler was unable to compile that benchmark successfully.

It is because of this poor performance of the GCC compiler on the *C-style* benchmarks that motivated us to work on a new compiler such as SUIF. We wanted the new compiler to generate efficient code for the *C-style* benchmarks, thereby eliminating the constraints of the 'DSP-style'. We will see how the SUIF-generated code performs in the subsequent sections.

6.2.2 SUIF Compiler – Initial Results

The diamond symbols (◊) in Figure 6.4 show the relative execution time of both the *DSP-style with SWP* and *C-style* benchmarks when compiled with the SUIF compiler after DGEN first became available. For the sake of comparison, we have also included the performance of both versions of the benchmarks when compiled with the GCC-based compiler, which is shown by the cross symbols (**X**). Solid lines represent execution time of the *DSP-style with SWP* version whereas the dashed lines indicate execution time of the *C-style* version. We have followed this notation for all the graphs shown in this chapter. Note that this DGEN did not have any of the optimizations described in Section 3.4. We see that the relative execution time of the SUIF-generated code for the *DSP-style with SWP* version benchmarks varies from the best of about 1.0 for **`lmsfir_32_64`** to the worst of almost 4.0 for **`lpc`**. We also notice that the SUIF-generated code for the *C-style* benchmark runs poorly as compared to the GCC-generated code for the same benchmarks. However we also see that GCC-generated code for the *C-style* version runs faster than the SUIF-generated code for the *DSP-style with SWP* version for some of the benchmarks such as **`lpc`** and **`adpcm`**, which is an indication that the SUIF-generated code has inefficiencies that are unrelated to the 'DSP-style'. In general, the results of Figure 6.4 indicate that, without sophisticated compiler optimizations, performance is most affected by the style in which a program is written. This trend is seen more in kernel benchmarks than in application benchmarks.

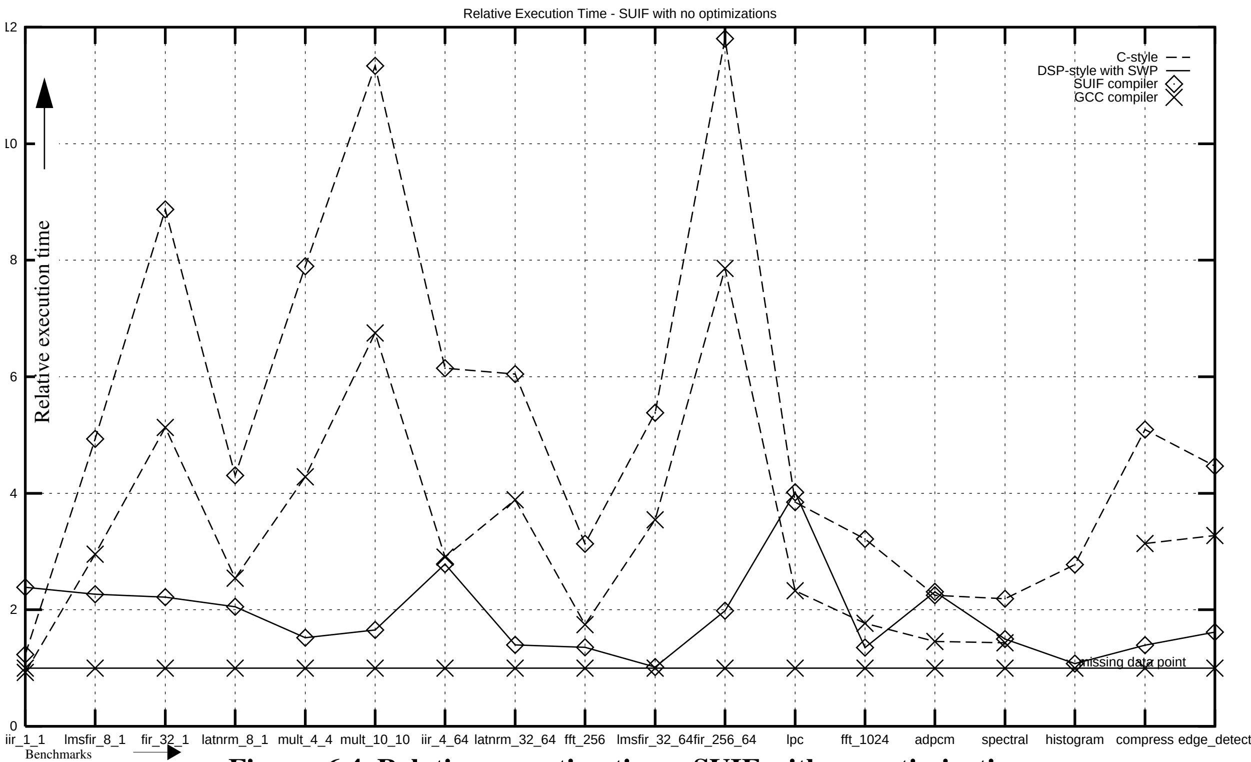

Figure 6.4 Relative execution time - SUIF with no optimizations

At this stage, we decided to investigate the differences between the SUIF-generated code and the GCC-generated code and identify those elements of the GCC-generated code that could be incorporated in the SUIF compiler. When we looked at unoptimized SUIF-generated code first, we noticed inefficiencies that needed improvements in the code generator DGEN.

6.2.3 Effect of unused label optimization

The first inefficiency we noticed in the SUIF-generated code was the presence of unused labels. This has been described in detail in Section 3.4.1. After we implemented this optimization, the performance of the SUIF-generated code improved considerably for the *C-style* benchmarks but only marginally for the *DSP-style with SWP* version benchmarks.

The graph in Figure 6.5 shows the relative execution time of the code with (indicated by the diamond symbol ◊) and without (indicated by the square symbol ❑) the unused label optimization for the *DSP-style with SWP* version benchmarks. The reason that the performance of the *DSP-style with SWP* version benchmarks does not improve noticeably with the unused label optimization is quite interesting. Most of the unused labels occur in the **`for`** loops. The *DSP-style with SWP* version benchmarks have their code written in the 'DSP-style', which means that the loops in their code already use induction variables instead of linear expressions involving the loop-index variable. Thus, it is easy for the post-optimizer in such cases to convert the **`for`** loops into hardware do-loops using the **`rep`**, **`do`** or **`do.d`** instructions of the DSP architecture. This conversion effectively removes the loop increment and loop test code that follows the **`continue`** label and as a result puts the **`continue`** label, **`break`** label and the do loop label together. See [Sin92] for an explanation of how the do-loop optimization works.

An example of this is shown in Figure 6.6. Figure 6.6(a) shows the code that SUIF generates for a **`for`** loop containing three labels **`LOOP_BEGIN`**, **`CONTINUE_LABEL`** and **`BREAK_LABEL`**. When this loop is converted into a hardware do-loop as shown in

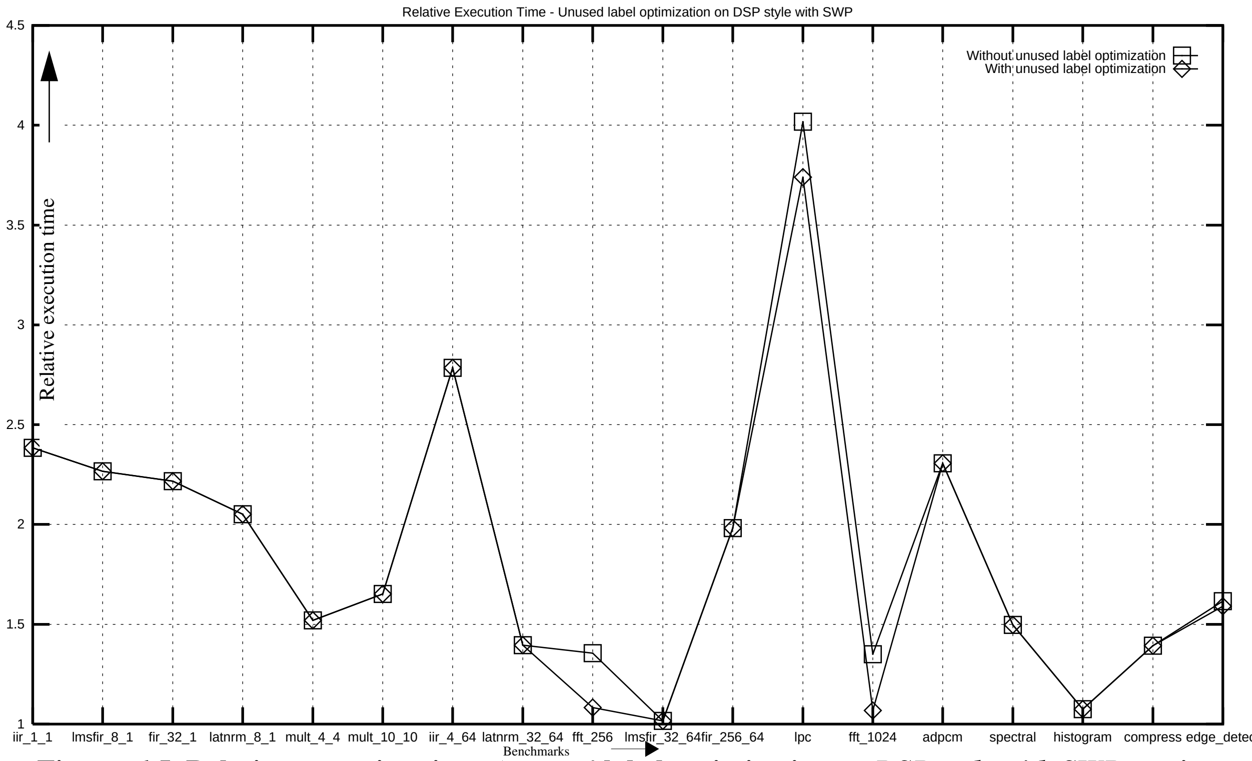

Figure 6.5 Relative execution time - Unused label optimization on *DSP-style with SWP* version

```
        movi.d  #1,d20
LOOP_BEGIN:
        ...
        ...
CONTINUE_LABEL:
        movi.d  #1,d19
        add.d   d20,d19,d20
        movi.d  #10,d8
        slt.d   d20,d8,d9
        bnez.d  d9,LOOP_BEGIN
BREAK_LABEL:
        ...
```

(a)

```
LOOP_BEGIN:
        do #10,LL1
        ...
        ...
CONTINUE_LABEL:
LL1:
BREAK_LABEL:
        ...
```

(b)

Figure 6.6 Example of hardware do loop conversion

Figure 6.6(b), we can see that both the **`CONTINUE_LABEL`** and **`BREAK_LABEL`** end up at the same place as the label **`LL1`** that is created by the post-optimizer for the **`do`** instruction. The label **`LOOP_BEGIN`** marks the place where the loop starts and because the **`do`** instruction itself marks the beginning of a new basic block, this label does not negatively affect performance by creating spurious basic blocks.

On the other hand, we can see the effect of unused label optimization on the *C-style* benchmarks in Figure 6.7. It shows significant improvement in the execution time of the *C-style* benchmarks by the use of unused label optimization. The hardware do-loop conversion is not performed for the *C-style* benchmarks, because many instructions inside the loop depend on the loop-index variable, and we have not used the INDVAR optimization yet.

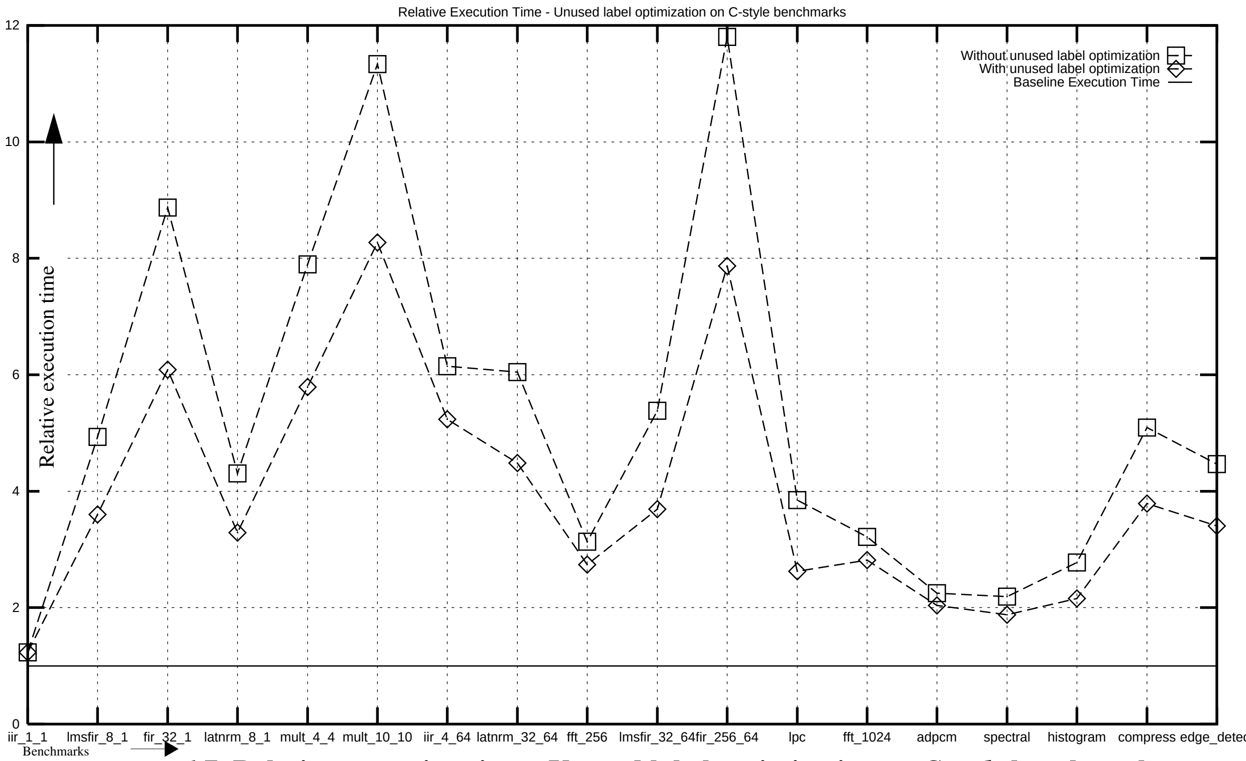

Figure 6.7 Relative execution time - Unused label optimization on *C-style* benchmarks

Consequently, the impact of the unused label optimization on the *C-style* benchmarks is significant after the elimination of the spurious `break` and `continue` labels in the `for` loops.

The lack of significant impact of the unused label optimization on the *DSP-style with SWP* version benchmarks shows how an optimization can interact with another in a complex fashion. In this case, the optimization of hardware do-loop generation works in such a way so as to make the unused label optimization almost redundant. It is not uncommon to find situations where two optimizations either aid each other or an earlier optimization in the compilation phase prevents the application of another useful optimization.

6.2.4 Effect of post-fix optimization

The second inefficiency we noticed in the code that SUIF generates is the translation of the post-increment and post-decrement operators. This has been described in detail in Section 3.4.2. Figure 6.8 shows the relative execution time of the *DSP-style with SWP* version benchmarks with and without the postfix optimization. We see a dramatic improvement in the performance of some of the benchmarks and, as explained in Section 3.4.2, that is because this optimization affects the code in more than one way. Note that this optimization had absolutely no effect on the performance of the *C-style* benchmarks, because these benchmarks do not use the post-increment or the post-decrement operators except in the third expression clause of the `for` statement, which the SUIF compiler treats differently from the normal post-increment or post-decrement operators[SUC94].

So far we have seen the impact of the optimizations we implemented in DGEN. From the next sub-section onwards we will mainly discuss the performance impact of the scalar optimizations used in the SUIF compiler. The performance graphs in these sub-sections represent the execution time of the SUIF-generated code with all the DGEN optimizations enabled.

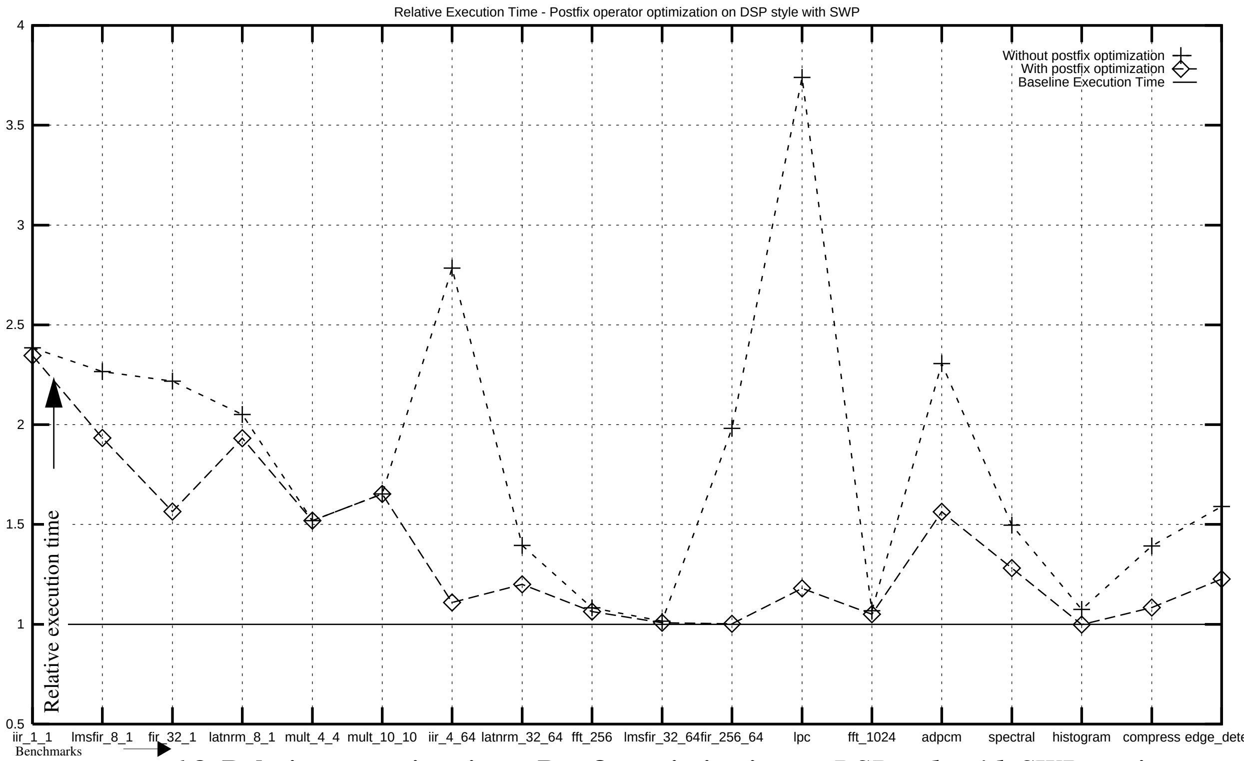

Figure 6.8 Relative execution time - Postfix optimization on *DSP-style with SWP* version

6.2.5 Using *oynk*: SUIF's scalar optimizer

The poor performance of the SUIF-generated code, even with improved DGEN, is not entirely unexpected because no optimization options were specified for the SUIF-compiler, whereas we used the full optimizations of the GCC-based compiler in our initial testing. A correct comparison requires using optimized code from both the compilers. At this stage we decided to use the optimization options **-O** and **-O2** of the SUIF compiler. Both **-O** and **-O2** options implement general scalar optimizations, although the **-O2** option runs an extra pass of code motion. However the result of using either of these options was disappointing.

All of the SUIF compiler's scalar optimizations are implemented in a pass called ***oynk***. Whenever either **-O** or **-O2** is specified, the SUIF compiler runs ***oynk*** to perform the required optimizations. In Section 4.1.4, we have described the problems with using ***oynk***. As documented in [SUC94][SUC94], ***oynk*** has many bugs which cause incomplete or incorrect compilation of many of our benchmarks. We also found that, although ***oynk*** performed the optimizations we wanted in most of the cases, it did not find all the opportunities for the loop-invariant code motion or the induction variable creation optimization.

Nonetheless, we persisted in using ***oynk*** for our experiments and attempted fixing some of its problems by modifying some of SUIF's other phases. Figure 6.9 shows the relative execution time of the *C-style* benchmarks with and without the use of the ***oynk*** optimizer. We notice that there is a big improvement in the performance of many of the benchmarks indicating that we were identifying the appropriate optimizations for what we wanted to achieve. However we can also see four “no data” points on the graph indicating that the SUIF compiler, or rather the ***oynk*** phase of the SUIF compiler, failed to compile these benchmarks properly. Particularly bad is the fact that the “no data” points represent four out of the six application benchmarks in our suite. Even when ***oynk*** is able to generate correct code for the benchmarks, we notice that the relative execution time ranges approximately between 1.5 and 3.0. When we analyzed the code generated by the SUIF compiler in this case, we noticed some of the shortcomings of the ***oynk*** optimizer such as ***oynk***'s inability to

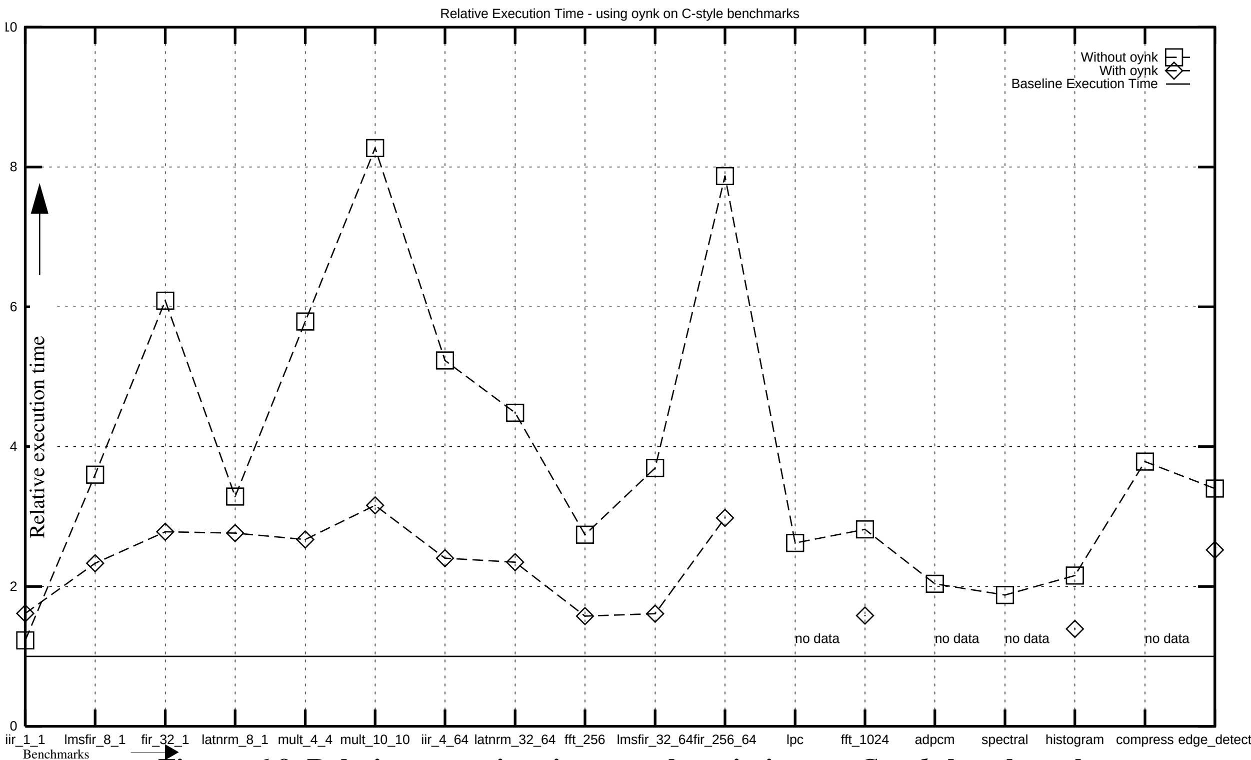

Figure 6.9 Relative execution time - oynk optimizer on *C-style* benchmarks

handle nested loops and complex expressions using loop-index variables of two or more enclosing loops. At this stage it became obvious that ***oynk*** was unusable for our work and no other scalar optimizer was available as part of the SUIF compiler system. After exploring all of our alternatives we decided to develop our own optimization pass to implement exactly those optimizations that we wanted. However, it was clear at this stage that the SUIF compiler in its available form was a long way off from helping us achieve our goal of being able to compile and run *C-style* benchmarks without sacrificing performance.

6.2.6 INDVAR: our own optimizer

Section 4.2 describes the main features of INDVAR, the optimizer we developed to implement the optimizations we wanted in the SUIF compiler. As described in Section 4.1, we expected the optimizations performed by INDVAR to improve the performance of the *C-style* benchmarks to a great extent. But how much improvement in the performance is possible? Figure 6.10 shows the impact of INDVAR on the execution time of the *C-style* benchmarks. This impact can be considered as truly remarkable. We can see that the relative execution time of the code optimized by INDVAR ranges from 1.0 to about 2.1. We can see up to four times improvement in the execution time of the code due to the optimizations performed by INDVAR. One thing to note here is that code that has been optimized by INDVAR can be further optimized by the post-optimizer using the hardware do-loop generation optimization. This is a second order benefit of using INDVAR. The cumulative effect of induction variable creation and the hardware do-loop generation produces a dramatic improvement in performance of the SUIF-generated code for the *C-style* benchmarks.

Let us now compare the performance of the code optimized by each of INDVAR, ***oynk*** and the GCC-based compiler for the *C-style* benchmarks. Figure 6.11 shows the relative execution times of the three different compiler-generated codes. First of all, notice that there are four missing data points in the case of the ***oynk*** graph and one missing data point in the case of the GCC graph. These indicate the failure of the respective compiler or the optimizer

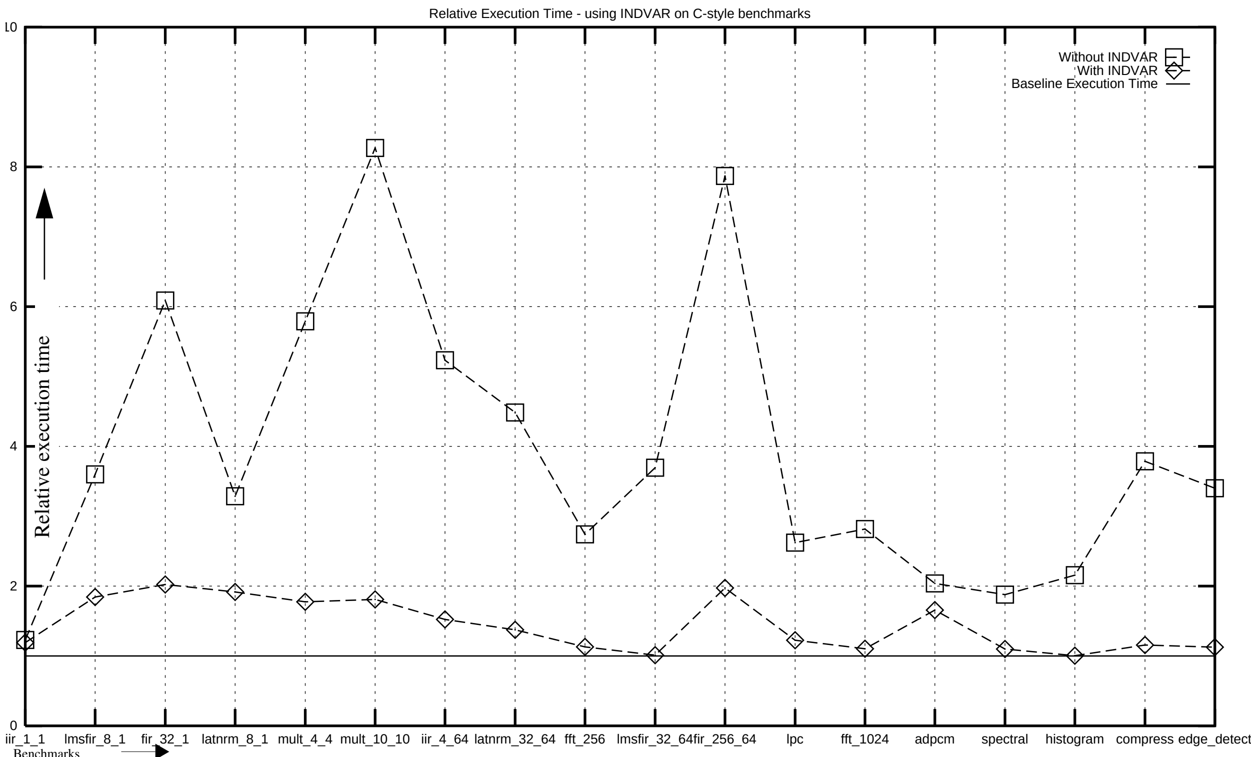

Figure 6.10 Relative execution time - INDVAR optimizer on *C-style* benchmarks

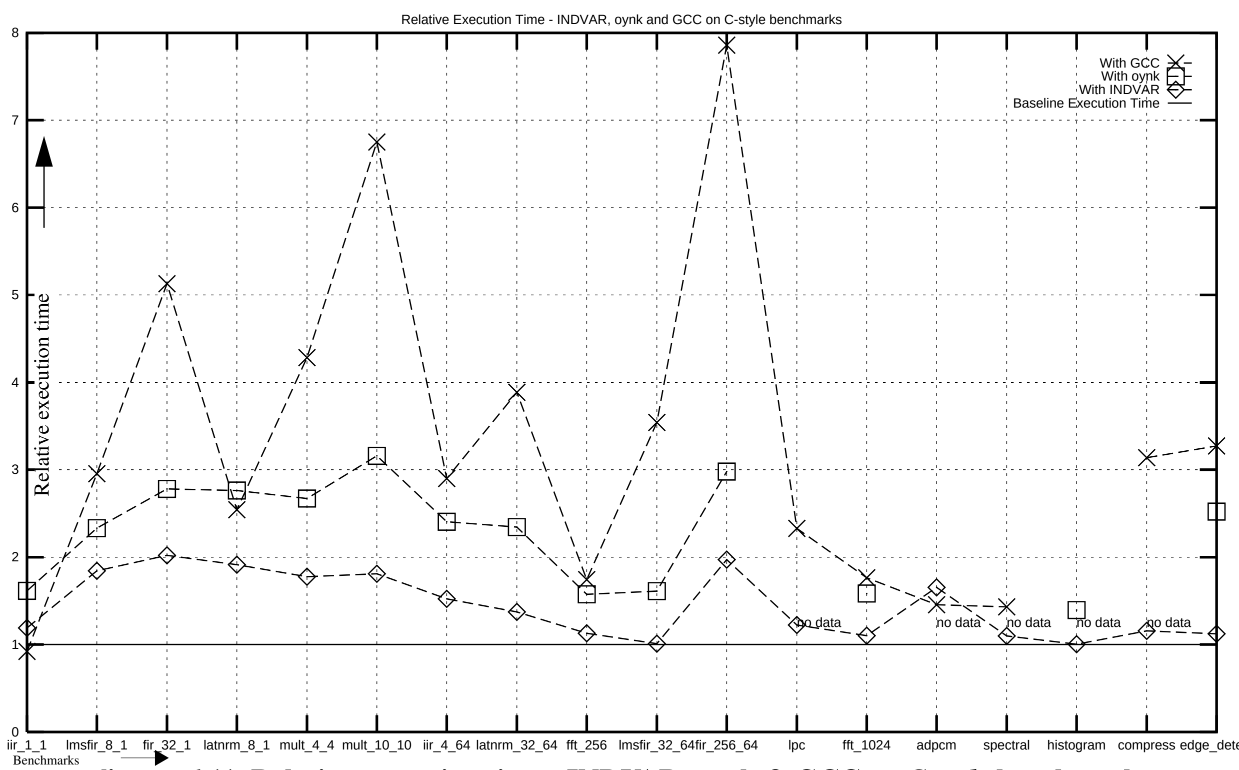

Figure 6.11 Relative execution time - INDVAR, oynk & GCC on *C-style* benchmarks

pass to compile those benchmarks. On the other hand we have no missing data points when using the INDVAR optimization. Execution time of the code optimized by INDVAR is the best for all benchmarks with the exception of **`iir_1_1`** and **`adpcm`** where the execution

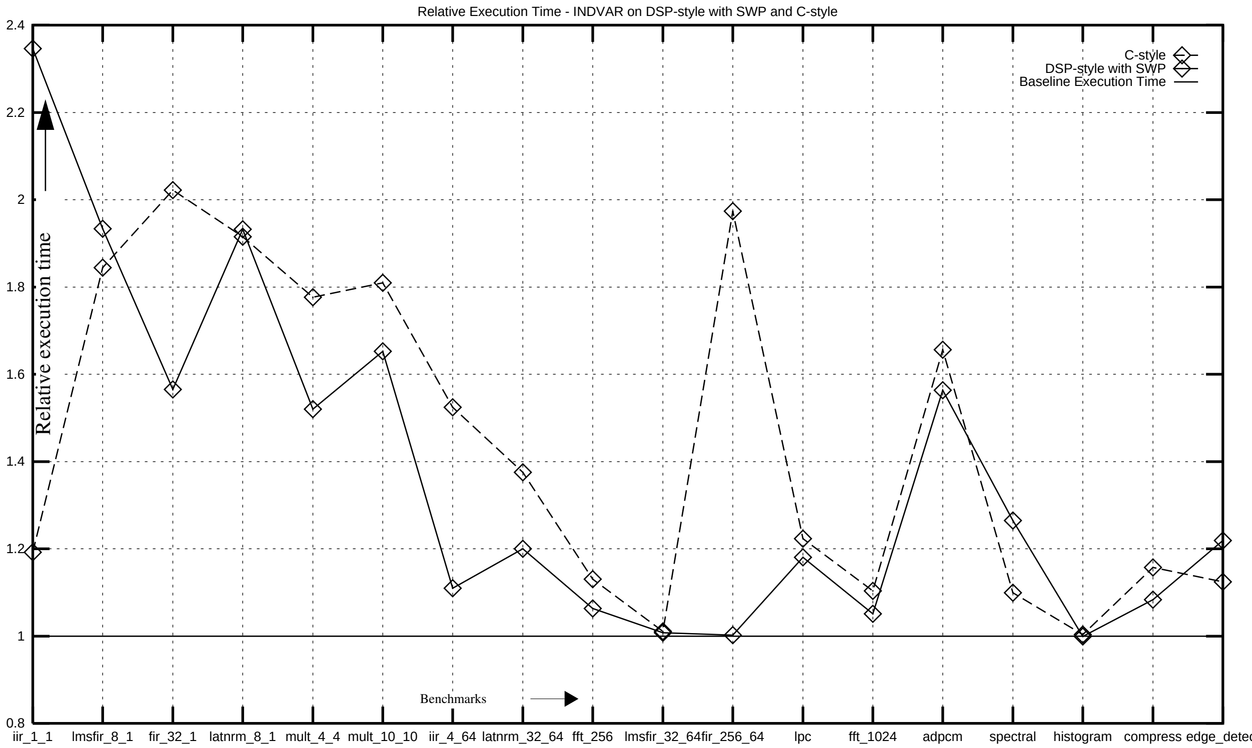

Figure 6.12 Relative execution time - INDVAR on *DSP-style with SWP* and *C-style* benchmarks

time of the GCC-generated code is slightly better than that of INDVAR-generated code.

Note that we are plotting the relative execution time using the baseline shown in Figure 6.1 which is the execution time of the GCC-compiled code for the *DSP-style with SWP* version benchmarks. Thus a relative execution time of 1.0 on the INDVAR graph would mean that the *C-style* benchmark when compiled with the SUIF + INDVAR combination is running as fast as the *DSP-style with SWP* benchmark compiled with the GCC-based compiler, which is the ideal performance we would like to achieve. Notice that we haven't yet reached that goal, with the exception of a few benchmarks such as **`lmsfir_32_64`** and **`histogram`**. There are two main reasons for this. First of all, the *DSP-style with SWP* version benchmarks have a hand-coded optimization called software pipelining which is absent in the *C-style* benchmarks. Secondly, the SUIF-generated code has many inefficiencies and overheads as compared to the GCC-generated code which we were unable to fix due to various reasons.

To eliminate the performance gap caused by the differences in the code generated by the two compilers and to understand better the performance impact of the different benchmark versions, we will compare the execution time of the code for the *DSP-style with SWP* version benchmarks with that for the *C-style* benchmarks when both are compiled using the SUIF compiler under identical conditions. Figure 6.12 shows these two execution times in a combined graph. We notice that, in four cases the *C-style* version benchmarks, surprisingly, are doing better than the *DSP-style with SWP* version, in three cases the execution time of the two versions is almost equal and in the eleven remaining cases, which is the majority, the *DSP-style with SWP* version benchmarks are doing better than their *C-style* counterparts. As mentioned earlier, the better performance of the *DSP-style with SWP* version can be attributed to the fact that these benchmarks have the hand-coded software pipelining optimization. If we modify the *C-style* benchmarks further to incorporate the same feature, will we then get the same performance as the *DSP-style with SWP* version benchmarks?

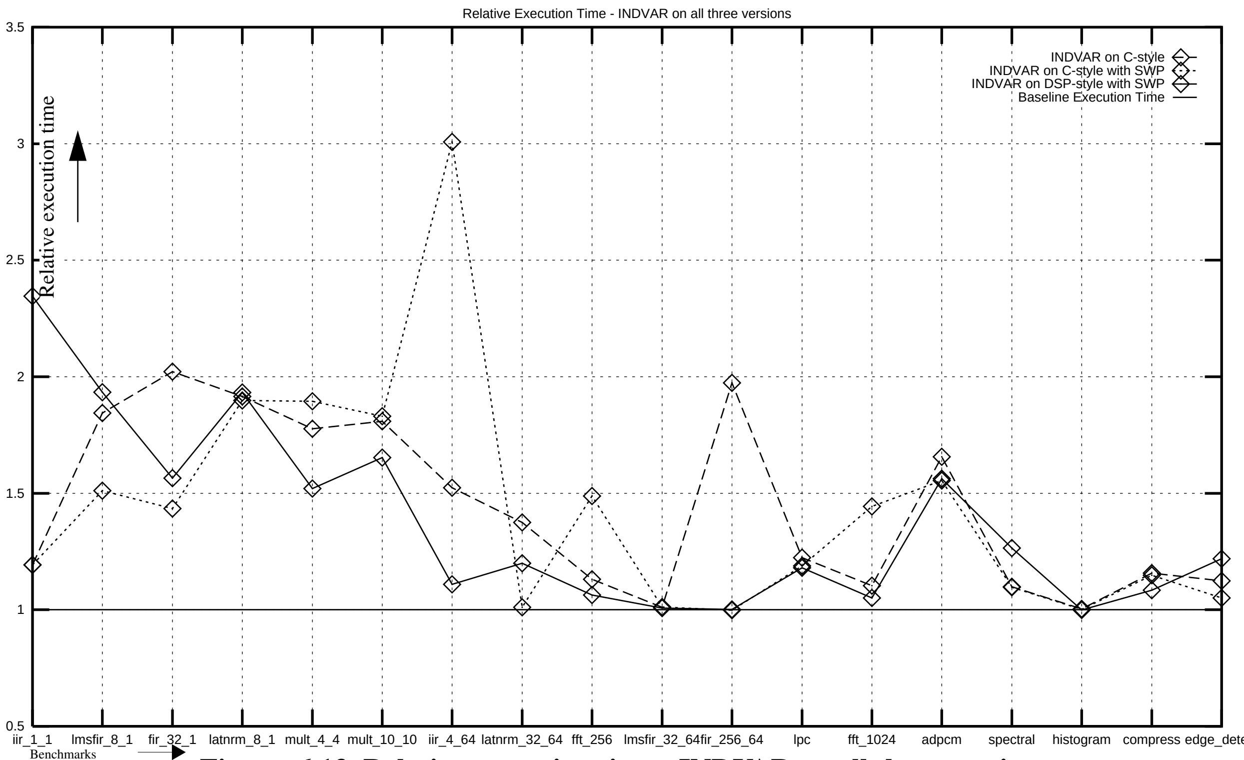

Figure 6.13 Relative execution time - INDVAR on all three versions

6.2.7 Effect of Software Pipelining

Since it was obvious that the hand-coded software pipelining optimization helped the *DSP-style with SWP* version benchmarks to a great extent, we decided to modify the *C-style* benchmarks further to create another version of benchmarks called *C-style with SWP*. These benchmarks are identical to the *C-style* ones except that all of the loops in the former are software pipelined in the same fashion as was done for the *DSP-style with SWP* version benchmarks. Figure 6.13 shows the relative execution time of the *C-style with SWP* benchmarks and that of the *C-style* benchmarks under identical conditions. What we see in Figure 6.13 can be aptly described as mixed results. There has been performance improvement in six benchmarks (maximum speedup of about 2.0 in **`fir_256_64`**), performance degradation in four benchmarks (maximum increase in execution time of about 100% in **`iir_4_64`**) and little or no difference in the remaining eight benchmarks. We could trace the performance degradation to an interesting factor that reinforces the significance of the hardware do-loop generation optimization performed by the post-optimizer.

Recall that INDVAR has a threshold or an upper limit on the number of new variables it creates while performing its optimization. When it reaches this threshold it stops further optimization for the current function. In such a situation, even if a loop is partially optimized, it would most probably have some linear expressions using the loop-index variable. The presence of these expressions inside the loop prevents the post-optimizer from converting the loop into a hardware do-loop as explained in Section 6.2.3. We already know the improvement in performance that results from hardware do-loop generation. What we noticed in a benchmark such as **`iir_4_64`** is that the *C-style* version could be completely optimized, whereas the *C-style with SWP* version could not be completely optimized due to the excess linear expressions inside loops. Note that if you have two nested loops, software pipelining the inner loop could result in more linear expressions for the outer loop due to the loop initialization and loop completion code that needs to be added around the inner loop. In short, the incomplete induction variable creation optimization resulted in performance degrading for the *C-style with SWP* version of **`iir_4_64`** because of the large second-order effect caused by the lack of hardware do-loop generation. Apart from demonstrating the importance of the hardware do-loop optimization, this also shows how two optimizations can interact with each other in complex ways.

Now that we have incorporated the software pipelining optimization in the *C-style* benchmarks, we can compare the execution time of the *C-style with SWP* benchmarks with the *DSP-style with SWP* version benchmarks. Figure 6.13 also shows the execution time of the *DSP-style with SWP* version benchmarks. Except for the spike at **`iir_4_64`**, the visual impression that we get from this graph is that the execution time of the *C-style with SWP* version of the benchmarks closely matches that of the *DSP-style with SWP* version benchmarks. Notice that the *C-style with SWP* version does better in six benchmarks, whereas it does worse in six benchmarks and in the remaining six there is little or no difference between the execution times of the two versions. As far as the spike at **`iir_4_64`** is concerned we can see from Figure 6.13 that the *C-style* benchmark has better performance than the *C-style with SWP* benchmark and software pipelining failed to improve performance because of register pressure and the resulting lack of hardware do loop optimization.

Another thing to note here is that in the absence of the software pipelining optimization in either of the front-end compilers or the back-end post-optimizer, we hand-coded this optimization in our source programs and created a *C-style with SWP* version of our benchmarks. If we had a compiler that implemented software pipelining optimization, we could have used it on the *C-style* benchmarks to try to get our ideal performance. It is interesting to note that this approach in fact would have benefitted us more because, as we noticed for the benchmark **`iir_4_64`**, the manual software pipelining adversely affected the induction variable creation optimization which in turn resulted into the hardware do-loop generation optimization not taking place. On the other hand, if software pipelining had been implemented as the last pass in the post-optimizer, we would have got the benefit of all three optimizations – induction variable creation, hardware do-loop generation and software pipelining – without the risk of one optimization adversely affecting another.

We decided to test our theory with an experimental version of a new post-optimizer that implements software-pipelining. We decided to compile all three versions of the benchmarks using the SUIF compiler and the new post-optimizer. Unfortunately the execution time of the code generated by the new post-optimizer without the software-pipelining optimization is different from the execution time of the code generated by the old post-optimizer, so the comparison between the relative execution time of the code generated by the new post-optimizer and that of the code generated by the old post-optimizer is somewhat difficult. Figure 6.14 shows the relative execution time of the *C-style* version benchmarks with and without the software-pipelining optimization of the post-optimizer. For the sake of comparison, it also shows the relative execution time of the *C-style with SWP* version benchmarks using the new post-optimizer without the software pipelining optimization enabled and the relative execution time of the same version using the old post-optimizer. If we ignore the graph corresponding to the old post-optimizer, we can see that the *C-style* version, when software pipelined by the post-optimizer has the best performance in all benchmarks. It never performs worse than the *C-style with SWP* version or the *C-style* version without the software-pipelining optimization. The ideal trend we expect to see is that the *C-style* version with the software pipelining optimization should have the best performance, followed by the *C-style with SWP* version followed by the *C-style* without the software pipelining optimization, when using the new post-optimizer. Remarkably, we can see this trend in thirteen out of the seventeen benchmarks. If we compare the execution time of the new post-optimizer with the old post-optimizer, we see that in six out of the seventeen benchmarks, the execution time of the code generated by the old post-optimizer for the *C-style with SWP* version is better than that of the *C-style* version generated by the new post-optimizer. Further investigation into the effects of the software pipelining optimization of the new post-optimizer on the performance of the *C-style* version benchmarks is warranted.

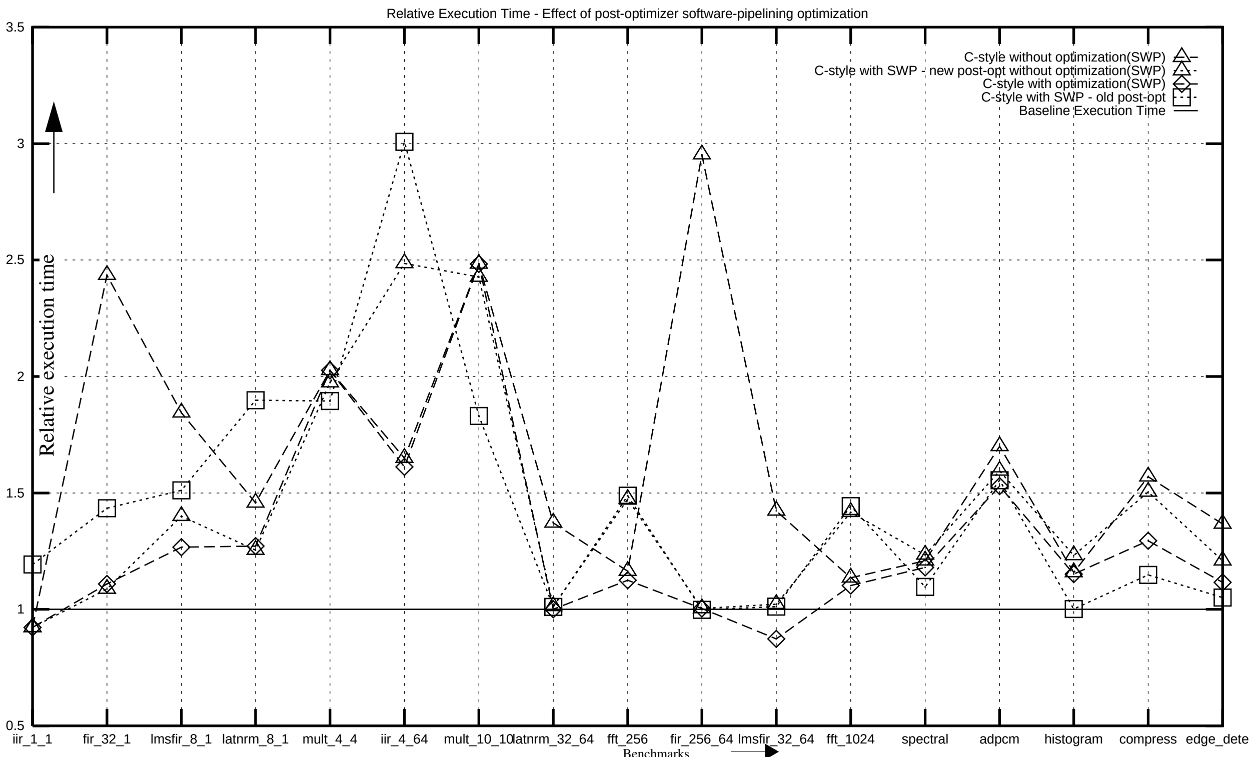

Figure 6.14 Relative execution time - Effect of post-optimizer software-pipelining optimization

6.3 Effect of ACSE

We described an optimization called common array sub-expression elimination (ACSE) in Section 4.4. This optimization seeks to eliminate redundant array accesses. When we investigated our benchmarks to select those which could offer opportunities to perform this optimization, we found five benchmarks. The *C-style* version of these five benchmarks were already written in such a way so as to avoid redundant array accesses. We modified these benchmarks further to create another version called *C-style-1* where the ACSE logic could find opportunities to perform the ACSE optimization. When we attempted this optimization on the *C-style-1* version of those five benchmarks, the results were mostly disappointing. Figure 6.15 shows the execution time of the *C-style-1* version of the benchmarks with and without the ACSE optimization. It also shows the execution time of the *C-style* version for the sake of comparison. We notice performance improvement in only one of the benchmarks, **`iir_4_64`**, while there was performance degradation in the other four when we compare the execution time of the ACSE optimized *C-style-1* version with the execution time of the *C-style* version. However we can see that the execution time of four *C-style-1* version benchmarks has improved due to the ACSE optimization.

When we investigated this further we found that the ACSE logic was unable to perform this optimization in most of those places where we had expected it to. This happened because the ACSE logic could not determine that certain array accesses were redundant, based on the information it had. On the other hand, a human programmer in such cases would know that these array accesses are redundant and would remove the redundant array accesses by doing the same optimization by hand. An example of this is shown in Figure 6.16. Notice that there are two read accesses to **`a[i]`**: one on line 1 and the other on line 3. There is also a write access occurring to **`a[i + result]`** on line 2. If the programmer who is writing this code knows that the value of the variable **`result`** can never be zero, then he knows that the second read access to **`a[i]`** on line 3 is redundant, and will remove it by copying the value obtained from the first read access to **`a[i]`** on line 1. However the ACSE logic does not have any information about the value of the variable

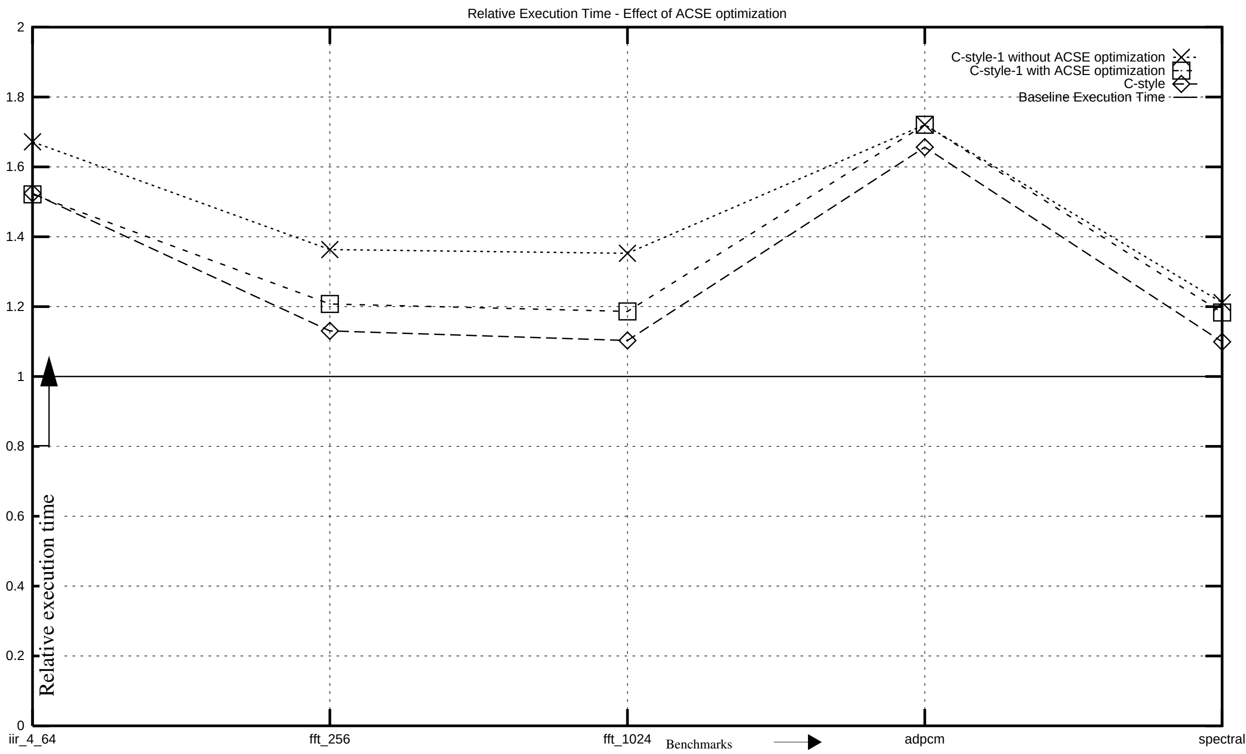

Figure 6.15 Relative execution time - Effect of ACSE optimization on *C-style-1* version

```
for (i = 0; i < N; i++) {
   x = a[i] * b[i];            /* line 1 */
   a[i + result] = x - b[i];   /* line 2 */
   b[i] = a[i];                /* line 3 */
}
```

Figure 6.16 Example of ACSE logic's failure to remove redundant array accesses

`result` and therefore has to make the safe assumption that a write to `a[i + result]` could change the value of `a[i]` that was read on line 1. In this case the ACSE logic would not find the second read access to `a[i]` on line 3 redundant and would not perform this optimization.

One might be tempted to think that more sophisticated data-flow analysis would help by providing information about the value of `result`. However , in most cases no amount of data-flow analysis can provide the precise information needed to perform the ACSE optimization. For example, in the above case, unless there is a very simple case of only one definition of `result` reaching line 2, it may not be possible for us to say whether the value of `result` can ever be zero or not.

Nonetheless it is possible to make a more aggressive implementation of this optimization. In the next chapter we suggest some ways to expand the scope of this optimization and make it more effective.

6.4 Summary

In previous chapters we described our efforts at retargeting the SUIF compiler to the DSP architecture and incorporating the new compiler into our development system. Our motivation was to build a compiler that would eliminate the constraints of the 'DSP-style'.

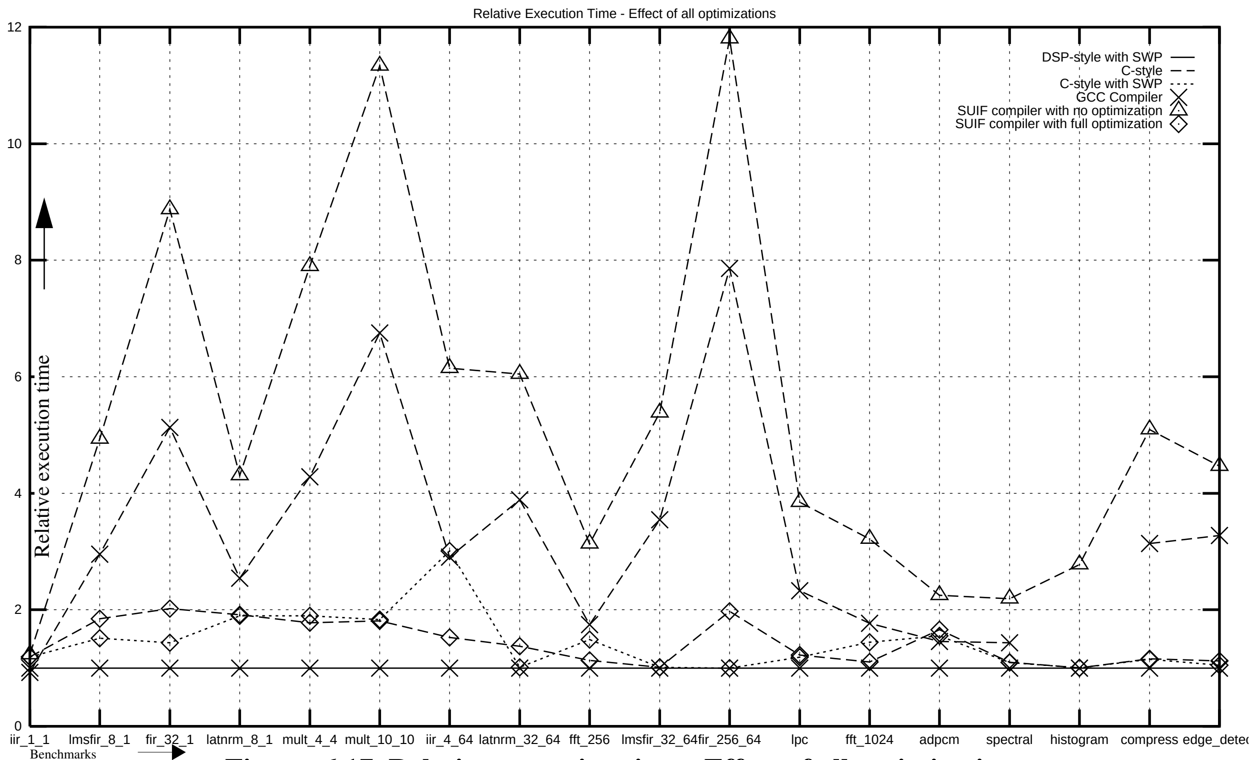

Figure 6.17 Relative execution time - Effect of all optimizations

The objective in the final phase of our work was to find out how well we did in this regard. We were equipped with two compilers and three versions of our benchmarks to assess the effectiveness of the optimizations we had built into the new compiler .

In this chapter we analyzed the performance of the code generated by the new compiler under different conditions which consisted of various combinations of benchmarks and optimizations. We also compared the performance we obtained from the SUIF compiler with the performance of the code generated by the existing GCC-based compiler we had. Figure 6.17 shows a graph that summarizes the execution time of the SUIF-generated code for both the *C-style* and *DSP-style with SWP* version benchmarks. For the sake of comparison, the execution time of the GCC-generated code for both versions is also included. Based on the comparisons we can conclude that we have achieved our goal of implementing a new optimizing compiler that would not sacrifice performance while at the same time remove the constraints of the DSP-style programming that our existing GCC-based compiler imposed upon application programmers.

However we also noticed some inefficiencies in the SUIF compiler and some scope for improvement in the optimizations we implemented. In the next chapter , we conclude this report by summarizing the work we did and by making suggestions for future work that would complement the work we have done.

Chapter 7 Conclusions

In this chapter we summarize the work we have done and present the main conclusions drawn from this work. Finally, we offer suggestions for areas that have potential for further work.

7.1 Summary and Conclusions

This work is based on earlier work that was done on a DSP architecture using a Very Long Instruction Word (VLIW) format and a customized optimizing C compiler for this architecture. This earlier work demonstrated the effectiveness of designing a specialized DSP architecture that is a good target for high-level language (HLL) compilers at the same time as developing an HLL compiler that makes effective use of the special features of the DSP architecture. The HLL compiler consisted of a GCC-based front-end compiler that translated C code into sequential assembly code and a post-optimizer that optimized this sequential code for the target DSP architecture. The post-optimizer performed optimizations specific to the DSP architecture such as low overhead loop generation, memory partitioning, instruction scheduling for the VLIW format, and (if required) modulo addressing.

The emphasis on machine-dependent optimizations demonstrated its effectiveness by generating good performance for a suite of kernel and application benchmarks. On the other hand, we could extract this performance out of the compile-generated code only by writing the benchmarks in a specialized 'DSP-style' that allowed greatest opportunities for optimization by the post-optimizer. The 'DSP-style' essentially consisted of constraints such as the use of pointer notation instead of array-index notation to access arrays. The pointer notation, in turn, would enable hardware do-loop optimization by the post-optimizer. It was the constraints of the 'DSP-style' that motivated us to explore the possibility of using a new front-end compiler with some powerful scalar optimizations that would eliminate the need for writing the source programs in the 'DSP-style'. We speculated that the front-end compiler, by employing suitable machine-independent scalar optimizations could essentially do the job of achieving 'DSP-style' performance from the input programs that were written in a more natural C style that allowed the use of array-index notation.

We looked at the SUIF compiler and decided to retarget it to our DSP architecture. We chose SUIF because of its flexible architecture, its support for dependence analysis, its support for annotations, and the fact that it is being used in on-going research on parallel and high-performance compilers. The first step in this process consisted of modifying the SUIF compiler's existing code generator, called MGEN, for the MIPS processor to make a code generator, that we call DGEN, for the DSP architecture. In our performance testing using the initial version of DGEN, we noticed that the SUIF-generated code for *C-style* benchmarks was between 2.0 to 12.0 times slower than the GCC-generated code for the *DSP-style with SWP* benchmarks. In DGEN itself, we found a few opportunities for optimizations. Some of these were related to the special features of the DSP architecture and others were optimizations necessitated by the inefficiencies we found in the SUIF code such as excess code for the post-fix operators.

After we got DGEN working properly, we concentrated on the scalar optimizations we wanted in the front-end compiler. We identified these to be induction variable creation or strength reduction inside loops and loop-invariant code motion. We tried using the SUIF compiler's optimization options to obtain these optimizations, but were not very successful in this because of limitations and bugs in ***oynk***, the only scalar optimizer available in the SUIF compiler. We then developed our own optimizer, called INDVAR, to implement exactly those optimizations that we wanted. The optimizations that INDVAR performed were induction variable creation, loop-invariant code motion, and common sub-expression elimination as a side effect of these two optimizations. Later on we also added an optimization called common array sub-expression elimination (ACSE) to eliminate redundant memory accesses inside a loop.

We also added a dependence analysis pass to the SUIF compiler that appended dependence and alias annotations to the SUIF intermediate code. The dependence annotations are used by the ACSE optimization described above and will be used by future versions of the post-optimizer to implement software pipelining. The alias annotations are used by both the post-optimizer and the DSP simulator to disambiguate memory references. In fact, it was the ability to attach annotations to the SUIF instructions, that was one of the reasons that we chose the SUIF compiler to retarget to the DSP architecture.

We also modified the post-optimizer front-end, so that it could read the SUIF annotations. We replaced the old parsing routines in the post-optimizer, which were written using the string searching and comparison functions of the C library, with a ***lex-*** and ***yacc-***based parser. This resulted in a well-structured parser that is based on a well-defined grammar and a lexical analyzer that is easy to maintain and modify.

When we were ready to test the SUIF compiler for performance, we created two more versions of our benchmarks. The original benchmarks, which we call *DSP-style with SWP* version, were written in the DSP-style and had software pipelined loops. We modified these to create the *C-style* version benchmarks, that were written in a more natural C style and did not use software-pipelining. The *C-style* version enabled us to assess the effectiveness of our optimization and compare the performance of the SUIF-generated code versus that of the GCC-generated code for programs that were not written in the 'DSP-style'. The *C-style with SWP* version was created from the *C-style* version of the benchmarks to compensate for the performance benefit that the *DSP-style with SWP* version of the benchmarks derived from their manually software pipelined loops.

Finally, we used the three versions of the benchmarks and the two compilers to collect data about the performance of the generated code under various conditions. The first conclusion that we can draw from this study is that the SUIF compiler was able to generate correct and efficient code for all three versions of the benchmarks with the optimizations that we had added to the SUIF compiler. Secondly, the SUIF-generated code for the *C-style* benchmarks was much more efficient than the GCC-generated code for the same version of the benchmarks, indicating that the SUIF compiler could compile source programs written in a more natural C style without sacrificing performance. In contrast, the GCC-generated code for the *DSP-style with SWP* version did much better than the SUIF-generated code for the same version of the benchmarks indicating that the GCC-based compiler performs certain optimizations that are missing in the SUIF compiler. The vast improvement in the performance of the *C-style* benchmarks brought about by the INDVAR optimizer validates our theory about the types of optimizations that are needed to eliminate the constraints of 'DSP-style' programming. Understandably, SUIF-generated or GCC-generated code for the *DSP-style with SWP* version of the benchmarks still executes faster than SUIF-generated code for the *C-style* version, because the *DSP-style with SWP* version used software pipelined loops. Using the *C-style with SWP* version to compensate for the lack of software pipelining optimization in our compiler, we could almost match the performance of the SUIF-generated code for the *DSP-style with SWP* version.

In conclusion, we were able to demonstrate that most elements of the 'DSP-style' could be eliminated by employing suitable machine-independent scalar optimizations in the front-end compiler that allowed the back-end compiler or the post-optimizer to perform its DSP-specific optimizations. The results of our performance testing showed that, for seven out of twelve kernel benchmarks and for five out of six application benchmarks, the performance of the *C-style with SWP* version optimized was equal to or better than the performance of the *DSP-style with SWP* version, when both versions are optimized by our compiler. This result indicates the effectiveness of our compiler in optimizing C-style code. However, we also observed a number of places in our implementation where improvements were possible. We discuss a number of these in the next section.

7.2 Future Directions

During the course of this project we encountered some issues that we would have liked to investigate further, but could not do so because of time constraints or because they were beyond the scope of this work. Some of these are suggestions for enhancements that can be made to the work we have done, while the others are issues that require additional work of a long term nature.

7.2.1 Future Directions: Short Term

We would like to suggest a number of improvements that can be made to the programs we developed or modified. We list them below individually for DGEN and the induction variable creation (IVC) optimization. In each list, the improvements that we believe might have the greatest impact on performance appear first.

DGEN

i) DGEN performs register allocation for node registers and pseudo registers. Apart from registers which are used for special purposes like parameter passing, the available machine registers are divided into two sets: *temporary* registers and *saved* registers. The set of *temporary* registers is used for allocating node registers whereas the set of *saved* registers is used for allocating pseudo registers. Currently the partition of registers into the two sets is hard-coded in DGEN. As an improvement we can make it dynamic in the third pass of DGEN, where DGEN can fix the size of the set of *temporary* registers depending on the maximum number of node registers that are live at any point inside a function. However this change may need more investigation, because reducing the number of *temporary* registers may adversely affect the ability of the post-optimizer to compact operations into instructions.

Induction Variable Creation

i) Currently the IVC logic does not create induction variables for trivially simple linear expressions including expressions consisting of the loop-index variable. However we can change the IVC logic to not suppress creation of induction variables for loop-index variables if there is room to create more variables as indicated by the threshold of variables. This would be very useful considering the fact that the hardware do-loop generation optimization requires elimination of all references to the loop-index variable inside the loop.

ii) On the other hand, we can suppress the loop-invariant code motion optimization for those constant expressions that involve no loop-index variables to make room for creating induction variables in situations where this enables the hardware do-loop optimization. This could be tricky in a single pass, since we cannot estimate how many variables are going to be needed by the IVC logic for the rest of the function that we are optimizing. However this could be integrated into a two-pass IVC logic that we have suggested in the next sub-section.

iii) An important improvement that can be made to the IVC logic is that it can avoid creating induction variables for those **`for`** loops which cannot be converted into a hardware do-loop by the post-optimizer, especially when the variables threshold prevents it from optimizing all loops. An example of a **`for`** loop that cannot be converted into a hardware do-loop is one where the number of iterations of that **`for`** loop are not known *a priori* at run time.

iv) We can eliminate the problem caused by the use of **`continue`** statement inside a **`for`** loop, as described in Section 4.2.4. Briefly, the solution consists of changing the destination address in the jump instructions, which resulted from the **`continue`** statement, to point to a new continue label.

7.2.2 Future Directions: Long Term

In this sub-section, we describe suggestions for future improvements or additions requiring work of a long term nature. Similar to the previous sub-section, our suggestions have been listed individually for DGEN, the induction variable creation (IVC) logic and the common array sub-expression elimination (ACSE) and each list starts with improvements that might have the greatest impact on performance.

<u>DGEN</u>

i) The register allocation in DGEN can be improved by using sophisticated algorithms such as graph coloring and splitting of variable live ranges.

ii) The possibility of using inter-procedural analysis (IPA) should be explored to implement interprocedural register allocation and generation of efficient header and trailer code.

<u>Induction Variable Creation</u>

i) We can make IVC a two-pass optimization. The first pass collects all the data pertaining to this optimization such as variable requirement for optimization of each **`for`** loop. The second pass actually carries out the optimization based on various criteria, some of which are described in the following points. The two-pass implementation of the IVC logic is a very important one, and it would allow many other improvements to be made.

ii) We can write a pass that converts **`while`** and **`do`**-**`while`** loops into **`for`** loops whenever possible, so that these **`for`** loops can be optimized by the IVC logic. This pass would precede the INDVAR pass in the compilation process.

iii) A significant improvement that can be included in the two-pass implementation of the IVC logic is that if a **`for`** loop cannot be completely optimized, it should be skipped altogether in favor of a different **`for`** loop that can be completely optimized. Because of the hardware do-loop generation optimization, we might get better mileage out of a complete IVC optimization of a **`for`** loop rather than a partial one when we have to choose between the two situations.

iv) In a two-pass implementation of IVC, the second pass can be designed so that it looks at the variable requirements of each **`for`** loop in a group of nested loops, and generates a linear program that can maximize the number of **`for`** loops to optimize. It can also attach different weights to each **`for`** loop to take into account factors such as the 'innerness' of a loop. Inner loops should be given higher weights, because of the greater benefit we get from optimizing inner loops.

v) Currently the post-optimizer is not able to handle spilled variables properly when performing the modulo-addressing pass. When this problem is fixed, we can explore the possibility of doing away with the variable threshold and allowing an infinite number of induction variables to be created. The effect of register spilling may not be as bad as we think.

vi) An ambitious addition that can be made to the IVC logic is a profile-driven optimization of `for` loops. If the IVC logic has to choose between different `for` loops to optimize, it could be designed to use data from a profiler as feedback on its previous optimization attempt. The successive iterations of code optimization and profile data gathering might yield the best optimization of the code.

Array Common Sub-expression Elimination

i) We can change this optimization also to be a two-pass optimization. The first pass would be able to generate a complete list of available expressions for a `for` loop and the second pass would be able to eliminate the redundant ones. We can combine this with the two-pass implementation of IVC.

ii) We can add data flow analysis of scalar variables to this pass to get more precise information on available array expressions.

7.3 Summary

What we can reasonably conclude so far is that using the optimizations we have implemented in the SUIF compiler, we have been able to match the performance of the *C-style with SWP* version benchmarks with that of the *DSP-style with SWP* version when both are compiled with the SUIF compiler. In most of these cases, however, the relative performance is still above 1.0. Recall that a relative performance of 1.0 for a benchmark means that the benchmark has exactly the same performance as its baseline performance, which is the performance of the *DSP-style with SWP* version of the benchmark compiled with the GCC-based compiler. Ideally we would like to obtain a relative performance of 1.0 for all of our benchmarks using our retargeted SUIF compiler. However, the SUIF compiler still has some inefficiencies which cause some performance degradation in the SUIF-generated code. In this chapter we suggested some optimizations and other solutions that could remedy this situation. At this time it is sufficient to say that these optimizations and solutions are orthogonal to the optimizations we have implemented in our work so far. If and when these optimizations are implemented, we can reasonably conclude that we will be able to bridge the performance gap that exists between the SUIF-generated code for the *C-style with SWP* version of the benchmarks and the GCC-generated code for the *DSP-style with SWP* version. When that happens we would be very close to the ideal situation of getting the relative performance of 1.0 for all of our *C-style with SWP* version of the benchmarks.

Appendix A

Instruction Formats

The following are the class definitions for all of the instruction formats that are used by DGEN. This is a listing of the file **`/pc/projects/dsp/base/dgen/asmInstr.h`**. The first **`enum`** definition identifies all the instruction formats that are defined by the respective class definitions. For example, the **`enum`** identifier **`aif_r4`** corresponds to the class **`ai_r4`**. The first class definition is that of the base class **`asmInstr`** and all the rest that follow it are derived from this class.

```
enum ai_formats {       /* assembly instruction formats */
    aif_xx,             /* pseudo-op and other free formats */
    aif_op,             /* only op-code (no operand) format */
    aif_rr,             /* rr format */
    aif_lr,             /* load from address register format */
    aif_sr,            /* store to address (in register) format */
    aif_r3,             /* rrr format */
    aif_r4,             /* rrrr format */
    aif_ri,             /* register immediate format */
    aif_ii,            /* single imm value format e.g. trap #x */
    aif_bj,             /* branch/jump to label format */
    aif_rj,             /* register immediate (label) format */
    aif_bi,             /* branch- condition reg- label format */
    aif_jr,             /* jump through register format */
    aif_ji              /* jump to label with count format */
};


/** Assembly language instruction definition */
class asmInstr {
    /* comment field information */
    int         nvals_c;              /* no. of icons in comment */
    icon_list_e *first_c, *last_c;  /* list of comment operands */

    /* annotation information */
    int         nvals_a;                   /* no. of annotation */
    annote      *first_a, *last_a;          /* list of annotes */

  protected:
    asmInstr(asmInstr *);
    void clear_lists() { nvals_c = 0; nvals_a = 0; first_c = NULL;
        last_c = NULL; first_a = NULL; last_a = NULL; }

    /* internal, specialized print routines for SUIF classes */
    void print_icon(icon ic, FILE *);

  public:
    /* comment field operations */
    int n_comments() { return nvals_c; }
    void appendCommentIcon(icon);
    icon popCommentIcon();

    /* annotation operations */
    int n_annotations() { return nvals_a; }
    void appendAnnote(annote *);
    void pushAnnote(annote *);
    annote *popAnnote();
    annote *getAnnote(char *);

    /* normal assembly instruction information */
    ai_ops      opcode;                 /* from AsmOp.h */
    ai_formats  format;                 /* instruction format */
    asmPath     *label;

    void setLabel(path_base *lp) { label = new asmPath(lp); }

    virtual ai_reg dest_reg();
    virtual ai_reg src1_reg();
    virtual ai_reg src2_reg();
    virtual ai_reg src3_reg();

    virtual ai_reg set_dest(ai_reg);
    virtual ai_reg set_src1(ai_reg);
    virtual ai_reg set_src2(ai_reg);
    virtual ai_reg set_src3(ai_reg);

    virtual asmInstr *copy();
    void read(ai_istream *);
    virtual void write(ai_ostream *);
    virtual void print(FILE *,int print_annotations = 1);
    void print_LabelAndOp(FILE *);
    void print_CommentAndAnnotes(FILE *);
    void debug_print(FILE *);

    asmInstr() { opcode = ao_nop; format = aif_xx; label = NULL;
        nvals_c = 0; nvals_a = 0; first_c = NULL; last_c = NULL;
        first_a = NULL; last_a = NULL; boosted_op = 0;
        dyn_nop = 0; }
    asmInstr(ai_ops o, ai_formats f) { opcode = o; format = f;
        label = NULL; nvals_c = 0; nvals_a = 0; first_c = NULL;
        last_c = NULL; first_a = NULL; last_a = NULL;
        boosted_op = 0; dyn_nop = 0; }
    virtual ~asmInstr() {}              /* destructor - aNDY */
};

/* subclass of aif_xx instructions */
struct ai_xx: asmInstr {
    /* pseudo-op instruction information */
    int         nvals_po;           /* no. of icons in pseudo-op */
    icon_list_e *first_po, *last_po; /* list of pseudo operands */

    /* pseudo-op instruction operations */
    void appendPseudoIcon(icon);
    icon popPseudoIcon();

    ai_xx():asmInstr(ao_nop,aif_xx)
        { nvals_po = 0; first_po = NULL; last_po = NULL; }
    ai_xx(ai_ops o):asmInstr(o,aif_xx)
        { nvals_po = 0; first_po = NULL; last_po = NULL; }
    ai_xx(ai_ops o, icon i):asmInstr(o,aif_xx)
        { nvals_po = 0; first_po = NULL; last_po = NULL;
          appendPseudoIcon(i); }
    ai_xx(ai_istream *is) { read(is); }
    ai_xx(ai_xx *);

    asmInstr * copy() { return new ai_xx(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_rr instructions */
struct ai_op: asmInstr {
    ai_op():asmInstr(ao_nop, aif_op) {}
    ai_op(ai_ops op):asmInstr(op, aif_op) {}
    ai_op(ai_istream *is) { read(is); }
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_rr instructions */
struct ai_rr: asmInstr {
    ai_reg      rd, rs1;      /* register source and destination */
    ai_reg dest_reg() { return rd; }
    ai_reg src1_reg() { return rs1; }
    ai_reg set_dest(ai_reg d) { rd = d; return d; }
    ai_reg set_src1(ai_reg s1) { rs1 = s1; return s1; }
```

```
    ai_rr(ai_formats xaif = aif_rr):asmInstr(ao_mov_d, xaif)
        { rd = ai_reg(); rs1 = ai_reg(); }
    ai_rr(ai_ops o, ai_formats xaif = aif_rr):asmInstr(o, xaif)
        { rd = ai_reg_dst(o);
  rs1 = ai_reg_src(o); }
    ai_rr(ai_ops o, int d, int s1, ai_formats xaif = aif_rr):as-
mInstr(o, xaif)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1,o); }
    ai_rr(ai_ops o, ai_reg d, ai_reg s1, ai_formats xaif =
aif_rr):
      asmInstr(o, xaif)
        { rd = d; rs1 = s1; }
    ai_rr(ai_ops o, ai_reg d, int s1, ai_formats xaif = aif_rr):
      asmInstr(o, xaif)
        { rd = d; rs1 = ai_reg_src(s1,o); }
    ai_rr(ai_istream *is) { read(is); }
    ai_rr(ai_rr *ai):asmInstr(ai) { rd = ai->rd; rs1 = ai->rs1; }

    asmInstr * copy() { return new ai_rr(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_lr instructions */
struct ai_lr : asmInstr {
    ai_reg      rd, rs1;      /* register source and destination */
    ai_reg dest_reg() { return rd; }
    ai_reg src1_reg() { return rs1; }
    ai_reg set_dest(ai_reg d) { rd = d; return d; }
    ai_reg set_src1(ai_reg s1) { rs1 = s1; return s1; }

    ai_lr():asmInstr(ao_ld_d, aif_lr)
    {
        rd = ai_reg(no_register, rt_general);
rs1 = ai_reg(no_register, rt_address);
    }
    ai_lr(ai_ops o):asmInstr(o, aif_lr)
    {
       rd = ai_reg_dst(o); rs1 = ai_reg(no_register, rt_address);
    }
    ai_lr(ai_ops o, int d, int s1):asmInstr(o, aif_lr)
    {
        rd = ai_reg_dst(d, o); rs1 = ai_reg(s1, rt_address);
    }
    ai_lr(ai_ops o, ai_reg d, int s1):asmInstr(o, aif_lr)
    {
        rd = d; rs1 = ai_reg(s1, rt_address);
        ASSERT(rd.type() == rt_from_op(o, opr_dest));
    }
    ai_lr(ai_ops o, ai_reg d, ai_reg s1):asmInstr(o, aif_lr)
    {
        rd = d; rs1 = s1;
        ASSERT(rd.type() == rt_from_op(o, opr_dest));
        ASSERT(rs1.type() == rt_address);
    }
    ai_lr(ai_istream *is) { read(is); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};
```



```
/* subclass of aif_sr instructions */
struct ai_sr : asmInstr {
    ai_reg      rs1, rs2;           /* register sources */
    ai_reg src1_reg() { return rs1; }
    ai_reg src2_reg() { return rs2; }
    ai_reg set_src1(ai_reg s) { rs1 = s; return s; }
    ai_reg set_src2(ai_reg s) { rs2 = s; return s; }

    ai_sr():asmInstr(ao_st_d, aif_sr)
    {
       rs1 = ai_reg(no_register, rt_address);
       rs2 = ai_reg(no_register, rt_general);
    }
    ai_sr(ai_ops o):asmInstr(o, aif_sr)
    {
       rs1 = ai_reg(no_register, rt_address);
       rs2 = ai_reg_src( o);
    }
    ai_sr(ai_ops o, int s1, int s2): asmInstr(o, aif_sr)
    {
       rs1 = ai_reg(s1, rt_address); rs2 = ai_reg_src(s2, o);
    }
    ai_sr(ai_ops o, ai_reg s1, int s2):asmInstr(o, aif_sr)
    {
       rs1 = s1; rs2 = ai_reg_src(s2, o);
    }
    ai_sr(ai_ops o, ai_reg s1, ai_reg s2):asmInstr(o, aif_sr)
    {
       rs1 = s1; rs2 = s2;
       ASSERT(rs2.type() == rt_from_op(o, opr_source));
    }
    ai_sr(ai_istream *is) { read(is); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_r3 instructions */
struct ai_r3: asmInstr {
    ai_reg      rd, rs1, rs2;           /* register sources and
destination */
    ai_reg dest_reg() { return rd; }
    ai_reg src1_reg() { return rs1; }
    ai_reg src2_reg() { return rs2; }
    ai_reg set_dest(ai_reg d) { rd = d; return d; }
    ai_reg set_src1(ai_reg s1) { rs1 = s1; return s1; }
    ai_reg set_src2(ai_reg s2) { rs2 = s2; return s2; }

    ai_r3():asmInstr(ao_add_d, aif_r3)
        { rd = ai_reg(); rs1 = ai_reg();
          rs2 = ai_reg(); }
    ai_r3(ai_ops o):asmInstr(o, aif_r3)
        { rd = ai_reg_dst( o); rs1 = ai_reg_src( o);
          rs2 = ai_reg_src( o); }
    ai_r3(ai_ops o, int d, int s1):asmInstr(o, aif_r3)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1, o);
          rs2 = ai_reg_src( o); }
    ai_r3(ai_ops o, int d, int s1, int s2):asmInstr(o, aif_r3)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1,o);
          rs2 = ai_reg_src(s2,o); }
    ai_r3(ai_ops o, ai_reg d, ai_reg s1):asmInstr(o, aif_r3)
        { rd = d; rs1 = s1; rs2 = ai_reg_src( o); }
    ai_r3(ai_ops o, ai_reg d, ai_reg s1, ai_reg s2):
```

```
        asmInstr(o, aif_r3)
        { rd = d; rs1 = s1; rs2 = s2; }
    ai_r3(ai_istream *is) { read(is); }
    ai_r3(ai_r3 *ai):asmInstr(ai) { rd = ai->rd; rs1 = ai->rs1;
                  rs2 = ai->rs2; }

    asmInstr * copy() { return new ai_r3(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_r4 instructions */
struct ai_r4: asmInstr {
    ai_reg      rd, rs1, rs2, rs3;  /* register sources and dest */
    ai_reg dest_reg() { return rd; }
    ai_reg src1_reg() { return rs1; }
    ai_reg src2_reg() { return rs2; }
    ai_reg src3_reg() { return rs3; }
    ai_reg set_dest(ai_reg d) { rd = d; return d; }
    ai_reg set_src1(ai_reg s1) { rs1 = s1; return s1; }
    ai_reg set_src2(ai_reg s2) { rs2 = s2; return s2; }
    ai_reg set_src3(ai_reg s3) { rs3 = s3; return s3; }

    ai_r4():asmInstr(ao_madd_d, aif_r4)
        { rd = ai_reg(); rs1 = ai_reg(); rs2 = ai_reg(); rs3 =
ai_reg(); }
    ai_r4(ai_ops o):asmInstr(o, aif_r4)
        { rd = ai_reg_dst( o); rs1 = ai_reg_src( o);
          rs2 = ai_reg_src( o); rs3 = ai_reg_src( o); }
    ai_r4(ai_ops o, int d, int s1):asmInstr(o, aif_r4)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1,o);
          rs2 = ai_reg_src( o); rs3 = ai_reg_src( o); }
    ai_r4(ai_ops o, int d, int s1, int s2):asmInstr(o, aif_r4)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1,o);
          rs2 = ai_reg_src(s2,o); rs3 = ai_reg_src( o); }
    ai_r4(ai_ops o, int d, int s1, int s2, int s3):asmInstr(o,
                 aif_r4)
        { rd = ai_reg_dst(d,o); rs1 = ai_reg_src(s1,o);
          rs2 = ai_reg_src(s2,o); rs3 = ai_reg_src(s3,o); }
    ai_r4(ai_ops o, ai_reg d, ai_reg s1):asmInstr(o, aif_r4)
        { rd = d; rs1 = s1; rs2 = ai_reg_src( o);
          rs3 = ai_reg_src( o); }
    ai_r4(ai_ops o, ai_reg d, ai_reg s1, ai_reg s2):asmInstr(o,
aif_r4)
        { rd = d; rs1 = s1; rs2 = s2; rs3 = ai_reg_src( o); }
    ai_r4(ai_ops o, ai_reg d, ai_reg s1, ai_reg s2, ai_reg s3):
         asmInstr(o, aif_r4)
        { rd = d; rs1 = s1; rs2 = s2; rs3 = s3; }
    ai_r4(ai_istream *is) { read(is); }
    ai_r4(ai_r4 *ai):asmInstr(ai) { rd = ai->rd; rs1 = ai->rs1;
                  rs2 = ai->rs2;
                  rs3 = ai->rs3; }

    asmInstr * copy() { return new ai_r4(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_ri instructions */
struct ai_ri: asmInstr {
    ai_reg      rd;             /* register destination */
    int         imm;            /* immediate value */
    ai_reg dest_reg() { return rd; }
    ai_reg set_dest(ai_reg d) { rd = d; return d; }
    int immed() { return imm; }

    ai_ri():asmInstr(ao_movi_d, aif_ri)
        { rd = ai_reg(); imm = 0; }
    ai_ri(ai_ops o):asmInstr(o, aif_ri)
        { rd = ai_reg_dst( o); imm = 0; }
    ai_ri(ai_ops o, int d, int i):asmInstr(o, aif_ri)
        { rd = ai_reg_dst(d,o); imm = i; }
    ai_ri(ai_ops o, ai_reg d, int i):asmInstr(o, aif_ri)
        { rd = d; imm = i; }
    ai_ri(ai_istream *is) { read(is); }
    ai_ri(ai_ri *ai):asmInstr(ai) { rd = ai->rd; imm = ai->imm; }

    asmInstr * copy() { return new ai_ri(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_ii instructions */
struct ai_ii: asmInstr {
    int         imm;            /* immediate value */
    int immed() { return imm; }

    ai_ii():asmInstr(ao_add_d, aif_ii)
        { imm = 0; }
    ai_ii(ai_ops o):asmInstr(o, aif_ii)
        { imm = 0; }
    ai_ii(ai_ops o, int i):asmInstr(o, aif_ii)
        { imm = i; }
    ai_ii(ai_istream *is) { read(is); }
    ai_ii(ai_ii *ai):asmInstr(ai) { imm = ai->imm; }

    asmInstr * copy() { return new ai_ii(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_bj instructions -
     this format always has symbolic target */
struct ai_bj: asmInstr {
    asmPath     *tgt;           /* target symbol */
    asmPath *target() { return tgt; }
    int offset() { return tgt->offset; }

    ai_bj(ai_formats xaif = aif_bj):asmInstr(ao_jmp, xaif)
        { tgt = NULL; }
    ai_bj(ai_ops o, ai_formats xaif = aif_bj):asmInstr(o, xaif)
        { tgt = NULL; }
    ai_bj(ai_ops o, path *t, ai_formats xaif = aif_bj):
         asmInstr(o, xaif)
        { tgt = new asmPath(t); }
    ai_bj(ai_istream *is) { read(is); }
    ai_bj(ai_bj *ai):asmInstr(ai)
        { tgt = ai->tgt; }

    asmInstr * copy() { return new ai_bj(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
```

```
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_bi instructions */
struct ai_bi: ai_bj {
    ai_reg      rs1;            /* register source */
    ai_reg src1_reg() { return rs1; }
    ai_reg set_src1(ai_reg s1) { rs1 = s1; return s1; }

    ai_bi():ai_bj(aif_bi) { rs1 = ai_reg(); }
    ai_bi(ai_ops o):ai_bj(o, aif_bi) { rs1 = ai_reg_src( o); }
    ai_bi(ai_ops o, int s1, path *t):ai_bj(o, t, aif_bi)
        { rs1 = ai_reg_src(s1,o); }
    ai_bi(ai_ops o, ai_reg s1, path *t):ai_bj(o, t, aif_bi)
        { rs1 = s1; }
    ai_bi(ai_istream *is) { read(is); }
    ai_bi(ai_bi *ai):ai_bj(ai)
        { rs1 = ai->rs1; }

    asmInstr * copy() { return new ai_bi(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_rj instructions */
struct ai_rj: ai_bj {
    ai_reg      rd;            /* register destination */
    ai_reg dest_reg() { return rd; }
    ai_reg set_dest(ai_reg r) { rd = r; return r; }

    ai_rj():ai_bj(aif_rj) { rd = ai_reg(); }
    ai_rj(ai_ops o):ai_bj(o, aif_rj) { rd = ai_reg_dst( o); }
    ai_rj(ai_ops o, int d, path *t):ai_bj(o, t, aif_rj)
        { rd = ai_reg(d,rt_address); }
    ai_rj(ai_ops o, ai_reg s1, path *t):ai_bj(o, t, aif_rj)
        { rd = s1; }
    ai_rj(ai_istream *is) { read(is); }
    ai_rj(ai_rj *ai):ai_bj(ai)
        { rd = ai->rd; }

    asmInstr * copy() { return new ai_rj(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};
/* subclass of aif_jr instructions */
struct ai_jr: asmInstr {
    ai_reg      rd;                /* register jump destination */
    ai_reg dest_reg() { return rd; }
    ai_reg set_dest(ai_reg s1) { rd = s1; return s1; }

    ai_jr():asmInstr(ao_jmp_a, aif_jr)
        { rd = ai_reg(no_register, rt_address); }
    ai_jr(ai_ops o):asmInstr(o, aif_jr)
        { rd = ai_reg(no_register, rt_address); }
    ai_jr(ai_ops o, int d):asmInstr(o, aif_jr)
        { rd = ai_reg(d,rt_address); }
    ai_jr(ai_istream *is) { read(is); }
    ai_jr(ai_jr *ai):asmInstr(ai) { rd = ai->rd; }

    asmInstr * copy() { return new ai_jr(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};

/* subclass of aif_ji instructions - integer and label */
struct ai_ji :ai_bj {
    int         imm;            /* immediate value */
    int immed() { return imm; }

    ai_ji():ai_bj(aif_ji) { imm = 0; }
    ai_ji(ai_ops o):ai_bj(o, aif_ji) { imm = 0; }
    ai_ji(ai_ops o, int i, path *t):ai_bj(o, t, aif_ji)
        { imm = i; }
    ai_ji(ai_istream *is) { read(is); }
    ai_ji(ai_ji *ai):ai_bj(ai)
        { imm = ai->imm; }

    asmInstr * copy() { return new ai_ji(this); }
    void read(ai_istream *);
    void write(ai_ostream *);
    void print(FILE *,int /* print annotations too */ = 1);
};
```

Appendix B

Post-optimizer Grammar

The following are the grammar productions that we defined for the new front-end of the post-optimizer .

```
goal  : program ;
program : program extern_def annotations |
          extern_def annotations ;
extern_def : function | asmdir | data_def | NL | error NL ;
data_def : LABELTYPE COLON NL | DATA_TYPE CONST NL ;
function : asmdir_begin function_body asmdir_end ;
asmdir_begin : DT_BEGIN LABELTYPE NL annotations ;
asmdir_end : DT_END LABELTYPE NL ;
function_body : function_body statement | function_body
            some_annotations | statement ;
statement     : asmdir | code_statement | NL ;
asmdir       : ASM_DIR_CODE operands NL | ASM_DIR_MISC_CODE oper-
ands NL ;
operands      : one_or_more_operands | <NULL> ;
one_or_more_operands : one_or_more_operands COMMA OPERAND |
             OPERAND ;
code_statement  : code_line ;

code_line      : LABELTYPE COLON NL | instruction NL ;
instruction    : zero_op_instruction | one_op_immed_instruction
               | one_op_indirect_reg_instruction
               | one_op_label_instruction
               | two_op_both_reg_instruction
               | two_op_first_indirect_second_reg_instruction
               | two_op_first_immed_second_reg_instruction
               | two_op_first_immed_second_label_instruction
               | two_op_first_reg_second_label_instruction
               | three_op_instruction | four_op_instruction ;
zero_op_instruction : OP_CODE ;
one_op_immed_instruction : OP_CODE POUND INTEGER ;
one_op_indirect_reg_instruction : OP_CODE LPAREN REG RPAREN ;
one_op_label_instruction : OP_CODE LABELTYPE ;
two_op_both_reg_instruction : OP_CODE REG COMMA REG ;
two_op_first_indirect_second_reg_instruction :  OP_CODE LPAREN
                  REG RPAREN COMMA REG ;
two_op_first_immed_second_reg_instruction : OP_CODE POUND INTEGER
                  COMMA REG
          | OP_CODE POUND LABELTYPE COMMA REG ;

two_op_first_immed_second_label_instruction : OP_CODE POUND
                  INTEGER COMMA LABELTYPE ;
two_op_first_reg_second_label_instruction : OP_CODE REG COMMA
```

```
                  LABELTYPE ;
three_op_instruction : OP_CODE REG COMMA REG COMMA REG ;
four_op_instruction : OP_CODE REG COMMA REG COMMA REG COMMA REG ;
annotations : some_annotations | <NULL> ;
some_annotations : some_annotations annotation_line
                  | annotation_line ;
annotation_line : ANNOTE_CHAR `[` ANNOTE_STRING
                  zero_or_more_annotes `]' NL ;
zero_or_more_annotes : zero_or_more_annotes annote_icon | <NULL>;
annote_icon  : ANNOTE_STRING ;
```

The following is a listing of the complete grammar specification for *yacc* that we developed for the post-optimizer. This can be found in the file **/pc/projects/dsp/base/optimizer/postopt.y**.

```
/* Grammar rules for the DSP assembly language output produced by
 * either of gcc-DSP or scc-DSP compilers. Compile using
 *
 *  yacc -d postopt.y
 *
 *
 */

%{

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include “instructions.h”
#include “optimizer.h”
#include “passes.h”
#define MIN_GAP  8
#define FORMAT_STRING_LEN 8

#define YYDEBUG 1
#define YYPRINT(file, type, value)   yyprint (file, type, value)

int linenum = 2;
extern int yylex();
extern int gcc_compiled;
int yyerror(char *s);
int match_opcode_format(enum instruction opcode, char *format);
int check_reg_type(enum instruction opcode, int position, int
reg);
void set_reg_use_def_bits(INSTRUCTION instruction_object);

/* The annotation system used is similar to SUIF’s.
 * BTW annote is a keyword in gcc !
 */

enum icon_discriminators {
    ic_int = ‘i’,
    ic_string = ‘s’,
    ic_path = ‘p’,
    ic_type = ‘t’,
    ic_register = ‘r’,
    ic_bottom = ‘b’
};

typedef struct icon {
    enum icon_discriminators discriminator;

    union {
        int ival;
        char *sval;
        char *pval;
        int tval;
        int rval;
    } val;
    struct icon *next;
    struct icon *prev;
} tIcon;

typedef struct sannote {
    char *name;
    struct icon *icon_list;
    struct sannote *next;
    struct sannote *prev;
} tAnnote;

tIcon *new_icon(enum icon_discriminators discriminator);
tAnnote *new_annote();

/* external function */
void attach_annotations(tAnnote *p);

%}

%union {
  OBJECT  object_token;
  enum instruction op_code;
  enum code asm_code;
  int int_token;
  float fp_token;
  char *text_token;
  struct icon *ic_token;
  struct sannote *an_token;
}

%token <int_token>  NL
%token <int_token>  COLON
%token <int_token>  COMMA
%token <int_token>  POUND
%token <int_token>  INTEGER
%token <int_token>  REG
%token <int_token>  LPAREN
%token <int_token>  RPAREN
%token <int_token>  ANNOTE_CHAR
%token <int_token> DT_BEGIN
%token <int_token> DT_END
%token <int_token> AN_TYPE_ID
%token <int_token> AN_REG
%token <text_token> LABELTYPE
%token <text_token> DATA_TYPE
%token <text_token> CONST
%token <text_token> OPERAND
%token <text_token> ASM_DIR_MISC_CODE
%token <text_token> ANNOTE_STRING
%token <text_token> AN_PATH
%token <op_code>    OP_CODE
%token <asm_code>   ASM_DIR_CODE

%type  <text_token>   operands
%type  <text_token>   one_or_more_operands
%type  <object_token> program
%type  <object_token> asmdir
%type  <object_token> data_def
%type  <object_token> asmdir_begin
%type  <object_token> asmdir_end
%type  <object_token> statement
%type  <object_token> code_statement
%type  <object_token> code_line
%type  <object_token> extern_def
%type  <object_token> function
%type  <object_token> function_body
%type  <object_token> instruction
%type  <object_token> zero_op_instruction
%type  <object_token> one_op_immed_instruction
%type  <object_token> one_op_indirect_reg_instruction
%type  <object_token> one_op_label_instruction
%type  <object_token> two_op_both_reg_instruction
```

```
%type  <object_token> two_op_first_indirect_second_re-
g_instruction
%type  <object_token> two_op_first_immed_second_reg_instruction
%type  <object_token> two_op_first_immed_second_label_instruction
%type  <object_token> two_op_first_reg_second_label_instruction
%type  <object_token> three_op_instruction
%type  <object_token> four_op_instruction
%type  <object_token> goal
%type  <ic_token> annote_icon
%type  <ic_token> zero_or_more_annotes
%type  <an_token> annotation_line
%type  <an_token> annotations
%type  <an_token> some_annotations

%start goal

%%


goal  : program
            {
                 OBJECT object;

                 assert(FIRST_OBJECT != NULL);
                 assert($1 == LAST_OBJECT);

                 /* Insert those first & last blank lines */
                 object = new_object();
                 PUT_CODE(object, NEW_LINE);

                 object->next = FIRST_OBJECT;
                 FIRST_OBJECT->previous = object;
                 FIRST_OBJECT = object;

                 object = new_object();
                 PUT_CODE(object, NEW_LINE);

                 object->previous = LAST_OBJECT;
                 LAST_OBJECT->next = object;
                 LAST_OBJECT = object;

                 $$ = $1;

            }
            ;

program : program extern_def annotations
            {
                 if ($2 != NULL) {
                    if ($1 != NULL)
                      $1->next = $2;
                    else {
                      /* if $1 is NULL,
                       * FIRST_OBJECT could not have been set */
                      assert(FIRST_OBJECT == NULL);
                      FIRST_OBJECT = $2;
                    }
                    $2->previous = $1;
                    while ($2->next != NULL) {
                      $2 = $2->next;
                    }
                    $$ = $2;
                    LAST_OBJECT = $2;
                 }
                 else
                    $$ = $1;

                 attach_annotations($3);
            }
        | extern_def annotations
            {
                 if ($1 != NULL) {
                    FIRST_OBJECT = $1;
                    while ($1->next != NULL) {
                      $1 = $1->next;
                    }
                    $$ = $1;
                    LAST_OBJECT = $1;
                 }
                 else
                   $$ = $1;

                 attach_annotations($2);
            }
            ;

extern_def : function
           | asmdir
           | data_def
           | NL
            {

                 OBJECT object = NULL;
                 if (gcc_compiled) {
                    object = new_object();
                    PUT_CODE(object, NEW_LINE);
                 }
                 $$ = object;
            }
           | error NL
               {
                   /* report error */
                   fprintf(stderr,
            "Error(%d): syntax error, line ignored.\n",linenum);
                   $$ = NULL;
               }
           ;

data_def : LABELTYPE COLON NL
               {
                   OBJECT object;
                   char *name;

                   name = (char *) malloc(strlen($1)+3);
                   strcpy(name, $1);
                   strcat(name, ":\n");
                   free($1);

                   object = new_object();
                   PUT_CODE(object, LABEL);
                   PUT_DATA_TEXT(object, name);
                   object->next = NULL;
                   object->previous = NULL;
                   $$ = object;
               }
         | DATA_TYPE CONST NL
```

```
                {
    /* We assume that the lexer when it returns DATA_TYPE token
       it reads the whole text line and splits it into DATA_TYPE
       and CONST as text tokens as semantic values of $1 and $2.
    */
                    OBJECT object;
                    char *name;

                    name = (char *) malloc(strlen($1)+strlen($2)+
                          MIN_GAP+2);
                    sprintf(name,"%-8s %s\n",$1,$2);
                    free($1);
                    free($2);

                    object = new_object();
                    PUT_CODE(object, DEFINE_CONST);
                    PUT_DATA_TEXT(object, name);
                    object->next = NULL;
                    object->previous = NULL;
                    $$ = object;
                }
                ;

function        : asmdir_begin function_body asmdir_end
                {
                    assert(GET_CODE($1) == FUNCTION_BEGIN);
                    assert(GET_CODE($3) == FUNCTION_END);
                   if (strcmp(GET_DATA_TEXT($1),GET_DATA_TEXT($3)))
                    {
                     /* give warning that names in .begin and
                      * .end are not same , that .begin name is
                      * assumed to be correct. */
                       char *name;

                       fprintf(stderr,
"Warning(%d): names in `.end' & `.begin' directives don't
match.\n", linenum);
                       fprintf(stderr,
      "Info(%d): name in `.begin' is %s\n", linenum,
                               GET_DATA_TEXT($1));
                       fprintf(stderr,
      "Info(%d): name in `.end' is %s\n", linenum,
                               GET_DATA_TEXT($3));
                       name = (char *) malloc(strlen(
                                GET_DATA_TEXT($1)));
                       strcpy(name, GET_DATA_TEXT($3));
                       free(GET_DATA_TEXT($3));
                       strcpy(name, GET_DATA_TEXT($1));
                       PUT_DATA_TEXT($3, name);

                    }
                    NEXT_OBJECT($2) = $3;
                    PREV_OBJECT($3) = $2;
                    while (PREV_OBJECT($2) != NULL) {
                      $2 = PREV_OBJECT($2);
                    }
                    NEXT_OBJECT($1) = $2;
                    PREV_OBJECT($2) = $1;
                    $$ = $1;
                }
                ;

asmdir_begin : DT_BEGIN LABELTYPE NL annotations
```

```
               {
                   OBJECT object;
                   char *name;

                   name = (char *) malloc(strlen($2)+2);
                   sprintf(name, "%s\n",$2);
                   free($2);

                   object = new_object();
                   PUT_CODE(object, FUNCTION_BEGIN);
                   PUT_DATA_TEXT(object, name);
                   NEXT_OBJECT(object) = NULL;
                   PREV_OBJECT(object) = NULL;
                   $$ = object;
                   attach_annotations($4);
               }
               ;

asmdir_end : DT_END LABELTYPE NL
               {
                   OBJECT object;
                   char *name;

                   name = (char *) malloc(strlen($2)+MIN_GAP+2);
                   sprintf(name, "%s\n",$2);
                   free($2);

                   object = new_object();
                   PUT_CODE(object, FUNCTION_END);
                   PUT_DATA_TEXT(object, name);
                   NEXT_OBJECT(object) = NULL;
                   PREV_OBJECT(object) = NULL;
                   $$ = object;
               }
               ;

function_body : function_body statement
               {
                   if ($1 != NULL)
                      NEXT_OBJECT($1) = $2;
                   if ($2 != NULL) {
                      PREV_OBJECT($2) = $1;
                      $$ = $2;
                   }
                   else
                      $$ = $1;
               }
              | function_body some_annotations
               {
                   attach_annotations($2);
                   $$ = $1;
               }
              | statement
              ;

statement     : asmdir
              | code_statement
              | NL
               {

                 OBJECT object = NULL;
                 if (gcc_compiled) {
                    object = new_object();
```

```
                PUT_CODE(object, NEW_LINE);
             }
             $$ = object;
           }
           ;

asmdir        : ASM_DIR_CODE operands NL
           {
               OBJECT object;
               char *name;

               if ($2 == NULL) $2 = “”;
               object = new_object();
               PUT_CODE(object, $1);
               name = (char *) malloc(strlen($2)+2);
               sprintf(name,”%s\n”,$2);
               if (*($2)) free($2);

               PUT_DATA_TEXT(object, name);
               NEXT_OBJECT(object) = NULL;
               PREV_OBJECT(object) = NULL;

               $$ = object;
           }
           | ASM_DIR_MISC_CODE operands NL
           {
               OBJECT object;
               char *name;

               if ($2 == NULL) $2 = “”;
               name = (char *) malloc(strlen($1)+strlen($2)+
                       MIN_GAP+2);
               sprintf(name, “%-*s %s\n”,MIN_GAP,$1,$2);
               free($1);
               if (*($2)) free($2);

               object = new_object();
               PUT_CODE(object, NULL_CODE);
               PUT_DATA_TEXT(object, name);
               NEXT_OBJECT(object) = NULL;
               PREV_OBJECT(object) = NULL;

               $$ = object;
           }
           ;

operands      : one_or_more_operands
              | { $$ = NULL; }
              ;

one_or_more_operands : one_or_more_operands COMMA OPERAND
           {
               char *name;

               if ($3 != NULL) {
                  name = (char *) malloc(strlen($1)+
                             strlen($3)+2);
                  sprintf(name,”%s,%s”,$1,$3);
                  free($1);
                  $$ = name;
               }
               else
                  $$ = $1;
           }
           | OPERAND
           {
               $$ = $1;
           }
           ;

code_statement  : code_line
               {
                   $$ = $1;
               }
               ;

code_line       : LABELTYPE COLON NL
               {
                   OBJECT object;
                   char *name;

                   name = (char *) malloc(strlen($1)+3);
                   strcpy(name, $1);
                   strcat(name, “:\n”);
                   free($1);

                   object = new_object();
                   PUT_CODE(object, LABEL);
                   PUT_DATA_TEXT(object, name);
                   object->next = NULL;
                   object->previous = NULL;
                   $$ = object;
               }
               | instruction NL  { $$ = $1; }
               ;

instruction     : zero_op_instruction
               | one_op_immed_instruction
               | one_op_indirect_reg_instruction
               | one_op_label_instruction
               | two_op_both_reg_instruction
               | two_op_first_indirect_second_reg_instruction
               | two_op_first_immed_second_reg_instruction
               | two_op_first_immed_second_label_instruction
               | two_op_first_reg_second_label_instruction
               | three_op_instruction
               | four_op_instruction
               ;

zero_op_instruction : OP_CODE
               {
                   OBJECT object;               /* New OBJECT */
                   INSTRUCTION instruction_object; /* New INSTR */

                   if (match_opcode_format($1, “x”)) {
                      object = new_object();
                      instruction_object = new_instruction();
                      PUT_CODE(object, OPCODE);
                      PUT_DATA_INSTR(object, instruction_object);
                      PUT_INSTR(instruction_object, $1);
                      $$ = object;
                   }
                   else
                      $$ = NULL;
               }
               ;
```

```
one_op_immed_instruction : OP_CODE POUND INTEGER
            {
               OBJECT object;              /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "l")) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_TYPE(instruction_object, 0, INT);
                  PUT_OP_INT(instruction_object, 0, $3);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

one_op_indirect_reg_instruction : OP_CODE LPAREN REG RPAREN
            {
               OBJECT object;                /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "A") &&
                  check_reg_type($1, 0, $3)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_INT(instruction_object, 0, $3);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

one_op_label_instruction : OP_CODE LABELTYPE
            {
               OBJECT object;                /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */
              char *label_or_symbol; /* Label/symbol buffer */

               if (match_opcode_format($1, "L")) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  label_or_symbol = $2;
                  PUT_OP_TYPE(instruction_object, 0, CHAR);
                  PUT_OP_CHAR(instruction_object, 0,
                      label_or_symbol);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

two_op_both_reg_instruction : OP_CODE REG COMMA REG
            {
               OBJECT object;                /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "rr") &&
                  check_reg_type($1, 0, $2) &&
                  check_reg_type($1, 1, $4)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_INT(instruction_object, 0, $2);
                  PUT_OP_INT(instruction_object, 1, $4);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

two_op_first_indirect_second_reg_instruction :  OP_CODE LPAREN
              REG RPAREN COMMA REG
            {
               OBJECT object;              /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "Ar") &&
                  check_reg_type($1, 0, $3) &&
                  check_reg_type($1, 1, $6)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_INT(instruction_object, 0, $3);
                  PUT_OP_INT(instruction_object, 1, $6);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

two_op_first_immed_second_reg_instruction : OP_CODE POUND INTEGER
              COMMA REG
            {
               OBJECT object;              /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "lr") &&
                  check_reg_type($1, 1, $5)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_TYPE(instruction_object, 0, INT);
```

```
                  PUT_OP_INT(instruction_object, 0, $3);
                  PUT_OP_INT(instruction_object, 1, $5);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            | OP_CODE POUND LABELTYPE COMMA REG
            {
               OBJECT object;               /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */
              char *label_or_symbol; /* Label/symbol buffer */

               if (match_opcode_format($1, "lr") &&
                  check_reg_type($1, 1, $5)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_TYPE(instruction_object, 0, CHAR);
                  label_or_symbol = $3;
                  PUT_OP_CHAR(instruction_object, 0,
                        label_or_symbol);
                  PUT_OP_INT(instruction_object, 1, $5);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

two_op_first_immed_second_label_instruction : OP_CODE POUND
                INTEGER COMMA LABELTYPE
            {
               OBJECT object;              /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */
              char *label_or_symbol; /* Label/symbol buffer */

               if (match_opcode_format($1, "lL")) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_TYPE(instruction_object, 0, INT);
                  PUT_OP_INT(instruction_object, 0, $3);
                  label_or_symbol = $5;
                  PUT_OP_TYPE(instruction_object, 1, CHAR);
                  PUT_OP_CHAR(instruction_object, 1,
                      label_or_symbol);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

two_op_first_reg_second_label_instruction : OP_CODE REG COMMA
                     LABELTYPE
            {
               OBJECT object;           /* New OBJECT */
               INSTRUCTION instruction_object;  /* New INSTR*/
               char *label_or_symbol;  /* Label/ symbol buf */

               if (match_opcode_format($1, "rL") &&
                  check_reg_type($1, 0, $2)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_INT(instruction_object, 0, $2);
                  label_or_symbol = $4;
                  PUT_OP_TYPE(instruction_object, 1, CHAR);
                  PUT_OP_CHAR(instruction_object, 1,
                       label_or_symbol);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

three_op_instruction : OP_CODE REG COMMA REG COMMA REG
            {
               OBJECT object;           /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "rrr") &&
                  check_reg_type($1, 0, $2) &&
                  check_reg_type($1, 1, $4) &&
                  check_reg_type($1, 2, $6)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
                  PUT_INSTR(instruction_object, $1);
                  PUT_OP_INT(instruction_object, 0, $2);
                  PUT_OP_INT(instruction_object, 1, $4);
                  PUT_OP_INT(instruction_object, 2, $6);
                  set_reg_use_def_bits(instruction_object);
                  $$ = object;
               }
               else
                  $$ = NULL;
            }
            ;

four_op_instruction : OP_CODE REG COMMA REG COMMA REG COMMA REG
            {
               OBJECT object;              /* New OBJECT */
               INSTRUCTION instruction_object; /* New INSTR */

               if (match_opcode_format($1, "rrrr") &&
                  check_reg_type($1, 0, $2) &&
                  check_reg_type($1, 1, $4) &&
                  check_reg_type($1, 2, $6) &&
                  check_reg_type($1, 3, $8)) {
                  object = new_object();
                  instruction_object = new_instruction();
                  PUT_CODE(object, OPCODE);
                  PUT_DATA_INSTR(object, instruction_object);
```

```
                        PUT_INSTR(instruction_object, $1);
                        PUT_OP_INT(instruction_object, 0, $2);
                        PUT_OP_INT(instruction_object, 1, $4);
                        PUT_OP_INT(instruction_object, 2, $6);
                        PUT_OP_INT(instruction_object, 3, $8);
                        set_reg_use_def_bits(instruction_object);
                        $$ = object;
                     }
                     else
                        $$ = NULL;
                  }
                  ;

annotations : some_annotations
            | { $$ = NULL; }
            ;

some_annotations : some_annotations annotation_line
               {
                 /* Value of $$ should be address of the last node
                  * in the linked list.
                  */
                 assert($1 != NULL);
                 $1->next = $2;
                 assert($2 != NULL);
                 $2->prev = $1;
                 $$ = $2;
               }
               | annotation_line
               ;

annotation_line : ANNOTE_CHAR `[` ANNOTE_STRING
                        zero_or_more_annotes `]' NL
               {
                 /* $4 points to the last node in the icon linked
                  * list.
                  */
                 tAnnote *temp = new_annote();
                 temp->name = $3;
                 /* traverse the linked list to come to the first
                  * node - ie. when prev is NULL.
                  */
                 while ($4 && $4->prev)
                    $4 = $4->prev;
                 temp->icon_list = $4;
                 $$ = temp;
               }
               ;

zero_or_more_annotes : zero_or_more_annotes annote_icon
               {
                 /* Value of $$ should be address of the last node
                  * in the linked list.
                  */
                 if ($1 != NULL) {
                    $1->next = $2;
                    if ($2 != NULL)
                       $2->prev = $1;
                 }
                 if ($2 != NULL)
                    $$ = $2;
                 else
                    $$ = $1;
               }
               | { $$ = NULL; }
               ;

annote_icon  : ANNOTE_STRING
               {
                  tIcon *temp;

                  temp = new_icon(ic_string);
                  temp->val.sval = $1;
                  $$ = temp;
               }
             | INTEGER
               {
                  tIcon *temp;

                  temp = new_icon(ic_int);
                  temp->val.ival = $1;
                  $$ = temp;
               }
             | AN_TYPE_ID
               {
                  tIcon *temp;

                  temp = new_icon(ic_type);
                  temp->val.tval = $1;
                  $$ = temp;
               }
             | AN_PATH
               {
                  tIcon *temp;

                  temp = new_icon(ic_path);
                  temp->val.pval = $1;
                  $$ = temp;
               }
             | AN_REG
               {
                  tIcon *temp;

                  temp = new_icon(ic_register);
                  temp->val.rval = $1;
                  $$ = temp;
               }
             | error
               {
                  fprintf(stderr,
    "Warning(%d): unrecognized annotation icon.\n",linenum);
                  $$ = NULL;
               }
             ;
%%


int yyerror(char *s)
{
    fprintf(stderr, "Error(%d): %s\n",linenum, s);
}

int match_opcode_format(enum instruction opcode, char *format)
{
   char new_format[FORMAT_STRING_LEN];
   int i;
```

```
   strcpy(new_format, INSTRUCTION_FORMAT(opcode));
   for (i = 0; new_format[i]; ++i) {
      if (new_format[i] == ‘d’ || new_format[i] == ‘f’ ||
             new_format[i] == ‘a’)
         new_format[i] = ‘r’;
      if (new_format[i] == ‘s’)
         new_format[i] = ‘l’;
   }
   if (strcmp(new_format, format)) {
      fprintf(stderr,
    “Error(%d): Invalid instruction format : %s.\n”,linenum,
                    INSTRUCTION_NAME(opcode));
      return 0;
   }
   return 1;
}

int check_reg_type(enum instruction opcode, int position, int reg)
{
   int retval = 0;
   if (reg < 0 && reg >= NUM_DATA_REG + NUM_ADDR_REG + NUM_FP_REG)
{
      fprintf(stderr,
“Error(%d): Invalid register specification.\n”,linenum);
      return 0;
   }
   switch (INSTRUCTION_FORMAT(opcode)[position]) {
      case ‘d’:         /* data register */
        retval = reg < NUM_DATA_REG;
        break;
      case ‘a’:         /* address register */
      case ‘A’:         /* indirect address register */
        retval = reg >= NUM_DATA_REG &&
               reg < NUM_DATA_REG + NUM_ADDR_REG;
        break;
      case ‘f’:         /* floating point register */
        retval = reg >= NUM_DATA_REG + NUM_ADDR_REG;
        break;
      default :
        assert(FALSE);   /* should not come here */
   }
   if (!retval)
      fprintf(stderr,
“Error(%d): \
    Operand %d register type incompatible with instruction.\n”,
              linenum,position);
   return retval;
}

void set_reg_use_def_bits(INSTRUCTION object)
{
  int operand_number;
  int operand_set;              /* Which operand is set */
  int type;                     /* Type of operand */
  int reg_or_immed;             /* Register number or immediate */
  int bit_vector_d;           /* Data register useage bit vector */
  int bit_vector_f;             /* FP register useage bit vector */
  int bit_vector_a;           /* Addr register usage bit vector */
  enum instruction instr;

  bit_vector_d = 0;
  bit_vector_f = 0;
  bit_vector_a = 0;
  instr = GET_INSTR(object);
  operand_set = INSTRUCTION_R_SET(instr);
  for (operand_number = 0;
       (type = INSTRUCTION_FORMAT(instr)[operand_number]);
                     ++operand_number) {
    reg_or_immed = GET_OP_INT(object, operand_number);
    switch (type) {
      case ‘d’:         /* data register */
          if (operand_set == operand_number)
            PUT_REG_SET(object, reg_or_immed);
          else {
            SET_BIT(bit_vector_d, reg_or_immed);
            PUT_REG_USE_D(object, bit_vector_d);
                /* should actually OR IT */
          }
          break;
      case ‘f’:         /* floating point register */
          if (operand_set == operand_number)
            PUT_REG_SET(object, reg_or_immed);
          else {
            SET_BIT(bit_vector_f,
               reg_or_immed - NUM_DATA_REG - NUM_ADDR_REG);
            PUT_REG_USE_F(object, bit_vector_f);
          }
          break;
      case ‘a’:         /* address register */
      case ‘A’:         /* indirect address register */
          if (operand_set == operand_number)
            PUT_REG_SET(object, reg_or_immed);
          else {
            SET_BIT(bit_vector_a, reg_or_immed - NUM_DATA_REG);
            PUT_REG_USE_A(object, bit_vector_a);
          }
          break;
      default:
          break;
    }
  }
}

tIcon *new_icon(enum icon_discriminators discriminator)
{
   tIcon *temp;

   temp = (tIcon *)malloc(sizeof(tIcon));
   if (!temp) assert(FALSE);
   temp->discriminator = discriminator;
   temp->next = NULL;
   temp->prev = NULL;
   return temp;
}

tAnnote *new_annote()
{
   tAnnote *temp;

   temp = (tAnnote *)malloc(sizeof(tAnnote));
   if (!temp) assert(FALSE);
   temp->icon_list = NULL;
   temp->next = NULL;
   temp->prev = NULL;
   return temp;
```

```
}

void destroy_icons(tIcon *p)
{
   /* note that p points to the first node in the linked list */
   if (p) {
     destroy_icons(p->next);
     if (p->discriminator == ic_string ||
         p->discriminator == ic_path) {
        free(p->val.sval);
     }
     free(p);
   }
}


static void yyprint (FILE *file,int type,YYSTYPE value)
{
   if (type == LABELTYPE)
     fprintf (file, “ %s”, value.text_token);
   else if (type == OPERAND)
     fprintf (file, “ %s”, value.text_token);
}
```

Appendix C

Lex Rules for the Post-optimizer Front-end

The following is the input specification that we developed for the new lexical analyzer for the post-optimizer . This specification can be found in the file **`/pc/projects/dsp/base/optimizer/postopt.l`**.

```
/* Scanner for the post-optimizer. Use flex (and not lex) to
 * compile this :
 *
 *  flex postopt.l
 *
 */

%{

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <stddef.h>


#include "instructions.h"
#include "optimizer.h"

struct icon;
struct sannote;

#include "y.tab.h"

extern int linenum;
int yywrap() { return 1; }

%}

COMMENT_CHAR     !
ANNOTATION_CHAR  %
WHITE_SPACE      [ \t]+

%x comment annote define asmdir

%%

<INITIAL,define,asmdir>{COMMENT_CHAR}            BEGIN(comment);
<comment>.*             /* ignore comments */
{ANNOTATION_CHAR}       BEGIN(annote); return ANNOTE_CHAR;
<INITIAL,asmdir,define,annote>{WHITE_SPACE} /* ignore white-sp */
<annote>\[              return '[';
<annote>\]              return ']';
<annote>\"[^\"]+\"      {
                         yylval.text_token = strdup(&yytext[1]);
                          /* skip " */
                          assert(yyleng >= 2);
                          assert(yylval.text_token[yyleng-2] ==
'"');
                          yylval.text_token[yyleng-2] = '\0';
                          /* remove " */
                          return ANNOTE_STRING;
                        }
<annote>\"type\([0-9]+\)\"  {
                          int temp;

                          assert(sscanf(yytext,"\"type(%d)\"",
                                 &temp) == 1);
                          yylval.int_token = temp;
                          return AN_TYPE_ID;
                        }
<annote>\<[^\>]+\>      {
                         yylval.text_token = strdup(&yytext[1]);
                          /* skip < */
                          assert(yyleng >= 2);
                          assert(yylval.text_token[yyleng-2] ==
'>');
                          yylval.text_token[yyleng-2] = '\0';
                          /* remove > */
                          return AN_PATH;
                        }
<annote>pr#[0-9]+       {
                          int temp;

                         assert(sscanf(yytext,"pr#%d", &temp) ==
                               1);
                          yylval.int_token = temp;
                          return AN_REG;
                        }
<define>[^ \t\n]+/\n  {

                              BEGIN(INITIAL);
                              yytext[yyleng] = 0;
                             yylval.text_token = strdup(yytext);
                              return CONST;
                        }
<asmdir>[-_a-zA-Z0-9]+   {

                              yytext[yyleng] = 0;
                             yylval.text_token = strdup(yytext);
                              return OPERAND;
                        }
<INITIAL,define,comment,annote,asmdir>"\n"     {
                                         BEGIN(INITIAL); ++linenum;
                                          return NL;
                        }
":"                     return COLON;
<INITIAL,asmdir>","     return COMMA;
"#"                     return POUND;
"("                     return LPAREN;
")"                     return RPAREN;

<INITIAL,annote>-?[0-9]+ { yytext[yyleng] = 0;
                          yylval.int_token = atoi(yytext);
                          return INTEGER;
                        }


[adf][0-9]{1,2}         {
                          int temp;
                          switch (yytext[0]) {
                            case 'd':
                              sscanf(yytext, "d%d", &temp);
                              yylval.int_token = 0;
                              break;
                            case 'a':
                              sscanf(yytext, "a%d", &temp);
                              yylval.int_token = NUM_DATA_REG;
                              break;
                            case 'f':
                              sscanf(yytext, "f%d", &temp);
                              yylval.int_token = NUM_DATA_REG +
                                NUM_ADDR_REG;
                              break;
                            default:
                            assert(0); /* should not come here */
                          }
                          assert(temp >= 0);
                          /* has to be positive */
```

```
            if (temp >= NUM_DATA_REG) {
               yylval.int_token = -1;
               /* parser will check */
            }
            else {
               yylval.int_token += temp;
            }
            return (REG);
          }

"ld.a"          yylval.op_code = LD_A; return OP_CODE;
"ld.d"          yylval.op_code = LD_D; return OP_CODE;
"ld.s"          yylval.op_code = LD_S; return OP_CODE;
"st.a"          yylval.op_code = ST_A; return OP_CODE;
"st.d"          yylval.op_code = ST_D; return OP_CODE;
"st.s"          yylval.op_code = ST_S; return OP_CODE;
"dec"           yylval.op_code = DEC; return OP_CODE;
"decmod"        yylval.op_code = DECMOD; return OP_CODE;
"decfft"        yylval.op_code = DECFFT; return OP_CODE;
"inc"           yylval.op_code = INC; return OP_CODE;
"incmod"        yylval.op_code = INCMOD; return OP_CODE;
"incfft"        yylval.op_code = INCFFT; return OP_CODE;
"and.a"         yylval.op_code = AND_A; return OP_CODE;
"asl.a"         yylval.op_code = ASL_A; return OP_CODE;
"asr.a"         yylval.op_code = ASR_A; return OP_CODE;
"ior.a"         yylval.op_code = IOR_A; return OP_CODE;
"lsl.a"         yylval.op_code = LSL_A; return OP_CODE;
"lsr.a"         yylval.op_code = LSR_A; return OP_CODE;
"mod.a"         yylval.op_code = MOD_A; return OP_CODE;
"mov.a"         yylval.op_code = MOV_A; return OP_CODE;
"movi.a"        yylval.op_code = MOVI_A; return OP_CODE;
"movs.a"        yylval.op_code = MOVS_A; return OP_CODE;
"movl.a"        yylval.op_code = MOVL_A; return OP_CODE;
"not.a"         yylval.op_code = NOT_A; return OP_CODE;
"xor.a"         yylval.op_code = XOR_A; return OP_CODE;
"seq.a"         yylval.op_code = SEQ_A; return OP_CODE;
"sne.a"         yylval.op_code = SNE_A; return OP_CODE;
"sgt.a"         yylval.op_code = SGT_A; return OP_CODE;
"slt.a"         yylval.op_code = SLT_A; return OP_CODE;
"add.d"         yylval.op_code = ADD_D; return OP_CODE;
"and.d"         yylval.op_code = AND_D; return OP_CODE;
"asl.d"         yylval.op_code = ASL_D; return OP_CODE;
"asr.d"         yylval.op_code = ASR_D; return OP_CODE;
"ior.d"         yylval.op_code = IOR_D; return OP_CODE;
"lsl.d"         yylval.op_code = LSL_D; return OP_CODE;
"lsr.d"         yylval.op_code = LSR_D; return OP_CODE;
"mod.d"         yylval.op_code = MOD_D; return OP_CODE;
"mov.d"         yylval.op_code = MOV_D; return OP_CODE;
"movi.d"        yylval.op_code = MOVI_D; return OP_CODE;
"movs.d"        yylval.op_code = MOVS_D; return OP_CODE;
"movl.d"        yylval.op_code = MOVL_D; return OP_CODE;
"not.d"         yylval.op_code = NOT_D; return OP_CODE;
"sub.d"         yylval.op_code = SUB_D; return OP_CODE;
"xor.d"         yylval.op_code = XOR_D; return OP_CODE;
"seq.d"         yylval.op_code = SEQ_D; return OP_CODE;
"sne.d"         yylval.op_code = SNE_D; return OP_CODE;
"sgt.d"         yylval.op_code = SGT_D; return OP_CODE;
"slt.d"         yylval.op_code = SLT_D; return OP_CODE;
"div.d"         yylval.op_code = DIV; return OP_CODE;
"madd"          yylval.op_code = MADD; return OP_CODE;
"msub"          yylval.op_code = MSUB; return OP_CODE;
"mult.d"        yylval.op_code = MULT; return OP_CODE;
"fadd.s"        yylval.op_code = FADD_S; return OP_CODE;
"fmod.s"        yylval.op_code = FMOD_S; return OP_CODE;
"fneg.s"        yylval.op_code = FNEG_S; return OP_CODE;
"fsub.s"        yylval.op_code = FSUB_S; return OP_CODE;
"fseq.s"        yylval.op_code = FSEQ_S; return OP_CODE;
"fsgt.s"        yylval.op_code = FSGT_S; return OP_CODE;
"fslt.s"        yylval.op_code = FSLT_S; return OP_CODE;
"fsne.s"        yylval.op_code = FSNE_S; return OP_CODE;
"fmov.s"        yylval.op_code = FMOV_S; return OP_CODE;
"fdiv.s"        yylval.op_code = FDIV_S; return OP_CODE;
"fmadd.s"       yylval.op_code = FMADD_S; return OP_CODE;
"fmsub.s"       yylval.op_code = FMSUB_S; return OP_CODE;
"fmult.s"       yylval.op_code = FMULT_S; return OP_CODE;
"do"            yylval.op_code = DO; return OP_CODE;
"do.a"          yylval.op_code = DO_A; return OP_CODE;
"do.d"          yylval.op_code = DO_D; return OP_CODE;
"enddo"         yylval.op_code = ENDDO; return OP_CODE;
"rep"           yylval.op_code = REP; return OP_CODE;
"beqz.a"        yylval.op_code = BEQZ_A; return OP_CODE;
"beqz.d"        yylval.op_code = BEQZ_D; return OP_CODE;
"beqz.s"        yylval.op_code = BEQZ_S; return OP_CODE;
"bnez.a"        yylval.op_code = BNEZ_A; return OP_CODE;
"bnez.d"        yylval.op_code = BNEZ_D; return OP_CODE;
"bnez.s"        yylval.op_code = BNEZ_S; return OP_CODE;
"jmp.a"         yylval.op_code = JMP_A; return OP_CODE;
"jmp"           yylval.op_code = JMP; return OP_CODE;
"jsr"           yylval.op_code = JSR; return OP_CODE;
"rts"           yylval.op_code = RTS; return OP_CODE;
"rti"           yylval.op_code = RTI; return OP_CODE;
"trap"          yylval.op_code = TRAP; return OP_CODE;
"movd2a"        yylval.op_code = D_TO_A; return OP_CODE;
"mova2d"        yylval.op_code = A_TO_D; return OP_CODE;
"movd2s"        yylval.op_code = D_TO_S; return OP_CODE;
"movs2d"        yylval.op_code = S_TO_D; return OP_CODE;
"nop"           yylval.op_code = NOP_PC; return OP_CODE;

".double"       |
".float"        |
".d_int"        |
".int"          |
".short"        |
".char"         |
".byte"         |
".addr"         {
                  BEGIN(define);
                  yytext[yyleng] = '\0';
                  yylval.text_token = strdup(yytext);
                  return (DATA_TYPE);
                }

".begin"        return DT_BEGIN;
".end"          return DT_END;
".text"         yylval.asm_code = TEXT; return ASM_-
DIR_CODE;
".data_X"       yylval.asm_code = DATA_X; return ASM_-
DIR_CODE;
".data_Y"       yylval.asm_code = DATA_Y; return ASM_-
DIR_CODE;
".skip"         yylval.asm_code = SKIP; return ASM_-
DIR_CODE;
".global"       BEGIN(asmdir); yylval.asm_code = GLOBAL;
return ASM_DIR_CODE;
```

```
“.common”                BEGIN(asmdir); yylval.asm_code = COMMON;
return ASM_DIR_CODE;

“.”[_a-zA-Z]+             {
                            BEGIN(asmdir);
                            yytext[yyleng] = ‘\0’;
                            yylval.text_token = strdup(yytext);
                            return ASM_DIR_MISC_CODE;
                          }


[_a-zA-Z][-_a-zA-Z0-9]*([\+\-][0-9]+)?/:?   {
                            yytext[yyleng] = ‘\0’;
                            yylval.text_token = strdup(yytext);
                            return (LABELTYPE);
                          }

%%
```

Bibliography

[ASU88] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[BGS93] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. *Compiler Transformations for High-Performance Computing*. University of California, Berkeley Technical Report No. UCB/CSD-93-781.

[Bis91] Bison on line documentation, GNU Project, Free Software Foundation, 1991.

[Cas94] Brian Case. Philips Hopes to Displace DSPs with VLIW. *Microprocessor Report, December 5, 1994* pages 12-15.

[DaJ94] Jack W. Davidson, Sanjay Jinturkar. Eliminating Redundant Memory Accesses. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1994*.

[EGK94] Kemal Ebcioglu, Randy D. Groves, Ki-Chang Kim, Gabriel M. Silberman & Isaac Ziv. VLIW Compilation Techniques in a Superscalar Environment. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1994*.

[GrM94] Torbjorn Granlund, Peter L. Montgomery. Division by Invariant Integers using Multiplication. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1994*.

[Hei94] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Technologies, Inc. 1994.

[HHN94] Joseph Hummel, Laurie Hendren, Alexandru Nicolau. A General Data Dependence Test for Dynamic, Pointer Based Data Structures. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1994*.

[LeE92] Ted G. Lewis, and Hesham El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, 1992.

[Mcf91] Scott McFarling. Procedure Merging with Instruction Caches. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1991* pages 71-79.

[PeH90] Karl Pettis, Robert Hansen. Profile Guided Code Positioning. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1990* pages 16-27.

[Per91] **`perl`** man pages. **`perl`** = Practical Extraction and Report Language.

[Sag93] Mazen A.R. Saghir. *Architectural and Compiler Support for DSP Applications*. M.A.Sc. Thesis. University of Toronto. 1993

[Sin92] Vijaya K. Singh. *An Optimizing C Compiler for a General Purpose DSP Architecture*. M.A.Sc. Thesis. University of Toronto. 1992

[Smp94] **`man`** pages for SUIF executables. Available in **`http://Suif.Stanford.edu`**. 1994.

[Str92] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1992.

[SUC94] SUIF Cookbook Documentation. Available from **`http://Suif.Stanford.edu`**. 1994

[SUD94] SUIF DEPENDENCE Library Documentation. Available from **`http://Suif.Stanford.edu`**. 1994

[SUL94] SUIF Library Documentation. Available from **`http://Suif.Stanford.edu`**. 1994

[SUM94] SUIF ReadMe File. Available from **`http://Suif.Stanford.edu`**. 1994

[SUR94] SUIF Release Notes. Available from **`http://Suif.Stanford.edu`**. 1994

[TjH92] Steven W.K. Tjiang, and John L. Hennessy. Sharlit - A Tool for Building Optimizers. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1992* pages 82-93.

[Tji93] Steven W.K. Tjiang. *Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers*. Ph. D. dissertation, Dept of Computer Science, Stanford University, July 1993.

[WoJ92] David B. Wortman, Michael D. Junkin. A Concurrent Compiler for Modula-2+. *ACM SIGPLAN Conference on Programming Language Design & Implementation, 1992* pages 68-81.