Debugging VLIW Code After Instruction Scheduling

by

Lyle Edward Cool
B.A., Whitworth College, 1980
B.S., University Of Portland, 1988

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the requirements for the degree
Master of Science
in
Computer Science and Engineering

July 1992

The thesis "Debugging VLIW Code After Instruction Scheduling" by Lyle Edward Cool has been examined and approved by the following Examination Committee:

Michael Wolfe, thesis advisor
Associate Professor

Steve Otto
Assistant Professor

Larry Meadows
The Portland Group, Incorporated

Acknowledgements

I learned a tremendous amount during the course of this work, and would like to express my gratitude for the opportunity to do so. I am grateful to Intel Corporation for maintaining a climate in which continuing education is encouraged. Specifically, Intel's tuition reimbursement program provided the funding for this research. Without that help, I probably would not have continued my education as long as I did. I am also grateful to Steve Otto and Larry Meadows for their very helpful feedback. I am especially grateful to Michael Wolfe for all of his help in completing this work. He gave innumerable suggestions, corrections, and comments on every phase of this project. He also gave me the encouragement to keep moving forward when I needed it most.

Table Of Contents

List Of Figures

Abstract

Debugging VLIW Code After Instruction Scheduling

by

Lyle Edward Cool

Supervising Professor: Michael Wolfe

There has been some work within the last ten years or so aimed at solving the problem of debugging optimized code for scalar architectures. Instruction scheduling for VLIW architectures causes additional debugging problems which have not been addressed at all. These additional problems include how to deal with the compaction of individual operations into VLIW words, and the transformations that occur as part of software pipelining and loop-unrolling. Earlier work on debugging optimized code is examined and modifications are proposed to handle these problems. Also, a way of presenting this information to the user in source-level terms is examined. The results indicate that effective source-level debugging of VLIW code after instruction scheduling is feasible.

CHAPTER 1.
Compiler Optimizations And Debugging

1.1. Introduction

Non-trivial programs rarely work correctly the first time. The cause of the errant behavior may be as simple as a misplaced semi-colon, or as fundamental as an incorrectly applied algorithm, or anything in between the two. When programs don't work correctly, they are said to contain *bugs*. The process of finding these bugs is called *debugging*.

One of the more primitive kinds of debugging is to insert statements at key places in the program to print out some portion of the state of the running program. *State* in this sense means the value of one or more variables at a particular point in the program. This kind of debugging is considered to be primitive for several reasons. First, the program must be modified, re-compiled, and re-executed every time the user wants to know more about the state of the running program. Second, the user cannot ordinarily control execution; that is, unless the program was specifically written to do so, it may not be possible to re-execute a specific portion with different data, or to skip a portion under some conditions. Third, the insertion of this extra code effectively creates a different program. In many cases, this does not cause a problem; in others, however, the presence of the extra debugging code may cause the bug to manifest itself in a different manner or not at all. Fortunately, there are software tools which are available to make the process of debugging more effective. They are called *debuggers*.

Debuggers are designed to display whatever might be of interest to the programmer. They also are designed to control execution such that a portion of a program can be executed, followed by a display or modification of items of interest, followed by more program execution, and so on. What constitutes "items of interest" will be different in different environments. An operating system, for example, manages the interface between high-level processes and low-level hardware. A debugger for an operating system should be able to start and stop processes; examine or modify the state of processes that are running; and so on. A person debugging a YACC program might want to step through the parsing of a language statement and to display the parse stack as it does so.

Perhaps surprisingly, not all programming language debuggers are necessarily similar in what they do. For example, stepping through an imperative-language program is not hard to imagine; however, stepping through a functional-language program may not make any sense. This thesis is aimed at programs written in imperative languages like C, Pascal, and Fortran.

1.2. Debuggers for Non-Optimized Imperative Programs.

Programs written in imperative languages consist of the steps that are required to achieve the purpose of the programmer, whatever that might be. These steps are in the form of a series of statements which do one of two things (broadly speaking). One class of statements is that which is concerned with the flow of execution. It includes such things as conditional and repetitive constructs like *if* or *while* statements. The other main class includes the statements that do the actual computations in the program. These statements do things like "get the value of a variable; multiply it by some number; store the new value into another variable". Bugs can creep into a program if the programmer inadvertently specifies the wrong series of statements, or the wrong variable as the destination for a computation. It could happen that the first nine statements do everything exactly right, but on the tenth, the result of a computation is stored into the wrong variable.

A debugger for an imperative language needs to be able to stop at any given place in the program so that the user can examine its state to see if anything has gone wrong yet, or to step through the program and watch as the state changes with each step. By allowing the user to control the program as it executes step by step, the debugger enables the user to find out at what point the computation goes awry.

The capabilities that are needed can be grouped into two broad categories that correspond closely with the characteristics that distinguish imperative language programs: the ability to get and set the current execution point, and the ability to get and set the value of a variable. In order to achieve these capabilities, the debugger must get some help from the architecture and from the compiler.

The architecture needs to have the ability to stop execution at any given address. Most often, this is accomplished indirectly by means of a special breakpoint opcode which causes a trap to the debugger. It can also be accomplished directly; that is, by having the ability to trap to the debugger when a previously specified address is reached during the execution. (Some architectures support both methods). It is also helpful, (but theoretically non-essential) to be able to trap to the debugger at the end of every instruction. This is frequently how "stepping" is achieved.

The compiler needs to provide to the debugger information that only it can collect during compilation. Only the compiler "knows", for example, which memory location corresponds to which variable. This *debug information* is created during compilation and typically consists of at least two maps: one contains the mapping from source statement to the address of the beginning of the corresponding block of object code; the other contains the mapping from each variable name to its location and type. These maps assume, of course, that each source statement *will* map to a distinct block of object code, and that each variable *will* map to a single location.

So, with some help from the architecture for controlling execution, and with some help from the compiler to provide mappings between object code and source code, effective debugging of non-optimized code can be readily achieved.

1.3. Some examples of debug information usage.

1.3.1. A Program Exception

Suppose that a program exception has occurred, and that the hardware (and/or the OS) has left the address of the offending instruction where the debugger can get it. In this situation, the debugger would consult the code map, discover the block of code from which the instruction came, and report back the corresponding statement as the source statement that was responsible for the exception.

1.3.2. Setting a Breakpoint

In the event that the user wants execution to stop at a particular statement, the debugger would consult the code map, find the address of the block of object code corresponding to the source statement, save the original opcode, and then place a breakpoint instruction at that location. Execution would then resume until the breakpoint was hit (or the program exited). In the event that the breakpoint was hit, an exception would be raised, causing the logic of the previous example to be followed.

1.3.3. Starting Execution Under User Control

Sometimes, a user might want to start (or resume) execution at a particular statement. In this situation, the debugger would consult the code map, find the address of the block of object which corresponds to that statement, and then set the instruction pointer to that location. This can be a very useful feature of a debugger, since it allows the user to skip some source lines, or repeat others, for example.

What if a user believes that the cause of a bug is that a source line increments a variable one too many times? To test that thesis without having to rebuild the program from scratch, the user can direct the debugger to skip that line, and then let the rest of the program run its course. If the output is correct when that line is skipped and incorrect otherwise, then the mystery is solved and can easily be fixed. If not, then the user can continue the search.

It should be noted that altering the flow of control of the program can easily have unintended consequences since assumptions made by the compiler when the code was generated may be violated.

1.3.4. Reporting the Value of a Program Variable

Assume that execution is momentarily stopped (perhaps due to a program exception like division by zero), and that the user wants to find out the value of variable "x". In this situation, the debugger would consult the data map, find the location of variable "x", retrieve its value, and then print it according to its type.

1.3.5. Changing the Value of a Program Variable

What if the user wants to change the value of variable "x" before execution resumes? In this situation, the debugger would consult the data map, find the location of variable "x", and change it to the desired value.

Suppose that the user believes that a variable somehow receives the wrong value, and that this incorrect value causes subsequent problems in the program. To test that thesis, the user could directly alter the value of the variable and then let the program continue, and then test the results.

Since the flow of control in a program can be dependent upon the value of certain variables, the same warning issued earlier (in the section "Starting Execution Under User Control) is relevant here as well. By altering such a variable, the program may very well take an unintended path.

1.4. How optimizations can cause problems for debuggers.

Optimizations can cause problems for debuggers because many optimizations violate assumptions made when the debug information was created. If a code motion optimization moves the evaluation of an expression to a different location, the conventional debug information may no longer be valid for that portion of the code. If the register allocation portion of a compiler decides to keep a variable in a register for the equivalent of several source statements, the debug information which indicates that the variable lives at a particular location in memory will be incorrect over those regions of the program. A debugger trying to use this invalid debug information would at best confuse the user and at worst give misleading or incorrect responses.

Following are some examples of optimizations which can cause problems for debuggers which rely on conventional debug information.

1.4.1. Constant Folding

The evaluation of a constant expression at compile-time (called constant folding) may be performed even when only the minimum optimization has been specified. For example, in

```
i = 2 * 3
i = i * 4
```

the compiler could easily determine that the value of "i" could safely be calculated statically. In other words, the compiler in essence replaced the original two statements with

```
i = 24
```

The user who expected to be able to "step" through the calculation at run time might be surprised that the calculation never happens.

1.4.2. Loop-invariant Code Motion

Loop-invariant code motion is an optimization in which an expression that is not a function of the loop iteration is moved outside of a loop. For example, in

```
for (i = 0; i < 10; i++)
{
      x = y * z;
      a[i] = i * x;
}
```

the assignment to "x" could safely be moved before the loop (assuming that a[i] was not aliased in some way to x, y, or z). By moving it out of the loop, the execution time is reduced since the expression will only be evaluated once rather than ten times. In effect, it is as if the user wrote

```
x = y * z;
for (i = 0; i < 10; i++)
{
      a[i] = i * x;
}
```

The user trying to debug the original source might be surprised that the assignment to x already happened upon entry to the loop.

1.4.3. Register Variables

Register promotion is another optimization that can cause problems. Consider the following source fragment:

```
int i;
for (i = 0; i < 10000; i++)
{
      a[i] = i * 2;
}
```

In the non-optimized version of this code, the value of `i` would have to be loaded from memory, used in the expression, incremented and stored back into memory for every iteration of the loop. If a debugger was used to query for the value of `i` half way through the execution of the loop, there would be no problem. The debug information would indicate the corresponding memory location from which the debugger would retrieve it.

During the optimization phase, however, the compiler might discover that it was a heavily used variable and therefore might decide to keep it in a register. Furthermore, if the loop represented the last use of the variable `i` the memory location might never get updated.

The problem is when the debugger uses the conventional debug information to retrieve the variable's value from what it thinks is the current location; it would not be aware that the memory location is temporarily non-current.

1.4.4. Instruction scheduling

Instruction scheduling creates problems because of the rearrangement of operations even across statement boundaries. In this sense, the problems are similar to those caused by other code motion optimizations. Since this thesis in fact concentrates on instruction scheduling, the specific problems created for debugging will be covered in detail subsequently.

1.5. Not all optimizations cause problems.

As long as an optimization does not violate the assumptions made when debug information was produced, it should create no problems for debuggers. Strength reduction, in which the compiler substitutes a "cheaper" set of instructions for the more obvious one suggested by the source code would not ordinarily present problems. For example, in

```
x = y * 2;
```

the compiler might substitute the presumably cheaper (but equivalent)

```
x = y << 1;
```

where "y << 1" means the value of y shifted left by one bit position.

1.6. Why we want to be able to debug optimized code.

There have been two common ways of dealing with the problem of debugging optimized code: turn off optimizations and re-compile if it is feasible to do so; or debug at the assembly language level if it is not. However, these workarounds may not always be acceptable. With regard to re-compiling with optimizations turned off: the bug may only show up in the presence of optimizations. One situation in which this can occur is when the compiler has an option to perform "unsafe" optimizations. For example:

```
1:    /* Source file a.c */
2:    int i1, i2;
3:    int *iptr = &i1;

1:    /* Source file b.c */
2:    extern int i1, i2, *iptr;
3:    main()
4:    {
5:            i1 = 1;
6:            *iptr = 2;
7:            i2 = i1;
8:    }
```

At the end of this program, `i1` and `i2` should equal 2. However, a compiler performing "unsafe" optimizations (under the direction of the user who "knew" what he was doing) might save `i1` in a register at line 5 and then use this old value on line 7. Another situation in which an optimized program might behave differently from a non-optimized one is when the compiler has bugs. It is unfortunately true that you cannot always assume that the compiler generates correct code. (In fact, compiler writers themselves obviously would want to be able to debug optimized code at the source-level in order to fix compiler bugs). It is also possible that speed and/or space considerations make debugging a non-optimized version impossible. With regard to debugging at the assembly language level: it could easily be tortuous for all but the most trivial programs. Therefore, the ability to have full source-level debugging of optimized code is very desirable.

CHAPTER 2.
Previous Work

2.1. "Debugging System Compatible With Optimizing Compiler" by Warren and Schlaeppi

This is what is called a "Technical Disclosure Bulletin" that was only a summary of the document I was really looking for [20]. That full document was called "Design of the FDS interactive debugging system" [26]. As far as I know, it is no longer available.

This is the earliest work I am aware of and yet surprisingly thorough. They recognized that the traditional debug information maps were not sufficient for debugging optimized code. They proposed a total of seven maps to handle the various situations which might arise in a debugging session:

- SLOC is a map from module, procedure, and statement to the corresponding object code.

- STYP is a map from module, procedure, and statement to the statement type (e.g., call, assignment, conditional, etc.), and variable. "Variable" means variable name in the case of an assignment statement; or the target in the case of a call statement, for example.

- VLOC is a map from module, procedure, statement, and variable to a list of locations. (A list is needed in the case where a variable lives at more than one location).

- VOPT is a map from module, procedure, statement, and variable to a table of "optimization information" which consists of the following: uses deleted, uses added, assignments deleted, assignments added, and liveness.

- ATTR is a map from module, procedure, and variable to the corresponding attributes. What they mean by attributes are whatever is needed to needed to address the variable. The example they give is a "based" variable which can be accessed only when the address of what it is "based on" is known.

- SDEL is a map from module and procedure to a list of deleted statements
- SUCC is a map from module, procedure, and statement to a list of possible successors to be used for single stepping.

Apparently, this project never got past the design stage, so there is no information on how practical it could be. It is clear, though, that a vast amount of debug information would have to be generated by the compiler and processed by the debugger for real-world programs.

2.2. "Symbolic Debugging of Optimized Code" by Hennessy

This paper presents an approach to detect when variables in the original source program are not *current* with respect to what is "really" happening in the optimized code. Dead-store elimination is an example: if the final store to a variable is deleted by the compiler because it is not needed later in the program, that variable is *non-current* from that point on.

This paper goes on to outline techniques by which the values of non-current variables can often be recovered. The recovery can be accomplished when other variables are available which can be used to calculate what the value of the non-current variable should be. The benefit to this approach is that the effect of the optimization can be hidden from the user. In other words, if the user queries the debugger for the value of a variable which happens to be non-current and recoverable, the debugger can perform the necessary calculations and report the result to the user as if the variable were current.

2.3. "Interactive Source-Level Debugging for Optimized Programs" by Zellweger

In the first four chapters of her Ph. D. dissertation, Zellweger gives a very nice overview of the issues surrounding the problems of debugging optimized code. She covers why optimizations cause problems; what we should expect in a debugger; specific optimizations and their effects on debugging; previous attempts to solve some of the problems; and many possible solution strategies. After this, in the main portion of her dissertation, she uses some of these possible strategies to enable a real-world debugger to handle two specific control-flow optimizations: procedure in-lining and tail merging (or cross jumping) which is shown below. Procedure in-lining is an optimization in which the body of a procedure is grafted into the source at each location where a call to the procedure originally existed. The primary benefit to this optimization is that it saves all the overhead of calling and returning from the procedure. (The down-side, of course, is that the program will be larger). Tail merging is an optimization in which several branches of code may contain identical "tails"; all of these except one are replaced by a jump to the one remaining tail. The benefit to this optimization is that it may save space, possibly allowing more of a program to fit into a cache.

Zellweger's solution for handling tail merging plays heavily in my work, so I will go into some detail about it. First, consider the following code fragment:

```
if (a)
     do "a" specific routine
     do generic routine
else if (b)
     do "b" specific routine
     do generic routine
else if (c)
     do "c" specific routine
     do generic routine
endif
```

After the tail merging optimization, the object code would resemble the following:

```
if (a)
     do "a" specific routine
     goto x
else if (b)
     do "b" specific routine
     goto x
else if (c)
     do "c" specific routine
x:
     do generic routine
endif
```

The problem that Zellweger attacked was to somehow maintain the illusion that lines 3, 6, and 9 map to distinct blocks of object code when in fact they all map to the same code. At the level of the original source, that should not present a problem. At the level of the actual object code, however, it is a big problem, since there is literally nothing in that block of code which would enable the debugger to determine if the code is executing on behalf of line 3, 6, or 9. In other words, if execution is temporarily halted in that block of code, the debugger would be unable to report which of the three lines was the current line. Zellweger's solution to this problem is *path determination*.

By examining the source above, it should be clear that if the current location is line 3, then control must have passed through line 2; if the current location is line 6, then control must have passed through line 5; and if the current location is line 9, then control must have passed through line 8. Therefore, it can be said that the code for lines 2, 5, and 8 is path-determining code for the (identical) object code executing on behalf of lines 3, 6, and 9. When we know which of the path determining lines executed most recently, then we know which of the several merged lines apparently executed next.

If the user asks the debugger to break in a region of code that has been transformed by the tail-merging optimization, then the debugger does some extra activity transparently. *Hidden* breakpoints are set at each instruction which is a path determiner for the merged region. Each time a hidden breakpoint is hit, a *time stamp* (a global debugger variable which is incremented before each use) is copied to a variable associated with that particular hidden breakpoint; then execution continues. When the user's breakpoint in the merged region is hit, the time stamp values of all the hidden breakpoints are examined. If the highest time stamp value was recorded for the line requested by the user (which indicates that it executed last), then execution halts, and the debugger reports that the user's breakpoint has been hit; otherwise, execution continues.

To illustrate, consider again the original source code above. Assume that the user requests a breakpoint at line 6 (which will cause a breakpoint instruction to be placed at label "x:" below. Here is the object code again, but this time with the potential background activity indicated within brackets:

```
if (a)
     do "a" specific routine
     [break; line_2_val = ++time_stamp; continue]
     goto x
else if (b)
     do "b" specific routine
     [break; line_5_val = ++time_stamp; continue]
     goto x
else if (c)
     do "c" specific routine
     [break; line_8_val = ++time_stamp; continue]
x:
     [
          break due to user-requested breakpoint;
          if line_5_val has highest value
               hand control to user;
          else
               ignore breakpoint; continue;
     ]
     do generic routine
endif
```

In Zellweger's solution, this activity is triggered only in response to a user's request. This brings up a major weakness in her solution (which she acknowledges): if the debugger gains control due to an unplanned activity such as an exception, then path determination will not ordinarily be active, meaning that the debugger will be unable to report which of several source lines is responsible for the problem.

Calling this *path determination* is something of a misnomer. After all, the debugger is not telling the user which path was taken through the code; it is only obtaining and using this information over a very limited region of the source in order to solve the problem caused by one particular optimization. A significant part of this work in fact advocates using essentially the same technique over the entire range of the program (or some smaller portion, if the user desires) for the purpose of reporting to the user which path was taken through the code. I will go into more detail in a later chapter.

2.4. "Recovery of Noncurrent Variables in Source-level Debugging of Optimized Code" by Srivastava

This work presents a corrected and improved version of the techniques presented in Hennessy's paper. In addition, techniques are presented to recover non-current variables from partial results available in temporaries and registers.

2.5. "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code" by Coutant, Meloy, Ruscetta

This contains a succinct explanation of the core problems in debugging optimized code: namely the code location problem, and the data value problem. The authors claim that

> "most of the problems turn out to be data value problems. Even when code is moved or eliminated, the main effect is that data values may not be as the user expects at every point during execution of the program."

Further on, they say that

> "Only those instructions that modify user data are troublesome, and only when they are moved across a statement boundary. Thus, many code motion problems can be solved by the same techniques as for data value problems."

The optimizations which were their primary targets were global register allocation, induction variable elimination, and instruction scheduling. The first two optimizations are fairly standard; the instruction scheduling that they dealt with was very specific to the target architecture. It had two purposes: to fill the branch-delay slots with useful instructions, and to avoid processor stalls on memory accesses.

Their solution to the problems created by these optimizations depended primarily on an extension to the conventional data map which they encapsulated in a "range" data structure. This structure has four fields in it: the first two fields give a low and a high value forming a "range" of instruction pointer values in which the next two fields apply. These specify in which memory location or register the variable lives, or its actual value if the variable has no location. (This last case might occur in the case of constant propagation, for example). By maintaining a table which maps a variable name to its location as a function of the instruction pointer, they are able to track the location of a variable as it changes through the program. Equally important is that they are able to detect when a variable has *no* location, and inform the user who is trying to get or set the value of that variable.

Instruction scheduling has the potential of re-ordering instructions without regard to the statement boundaries in the original source. In this case, the "range" data structure has a slightly different interpretation: it is to be viewed as the range of source lines in which the user needs to be warned that the variable in question had either an early or late update. For example, assume that a user requests a breakpoint on a statement containing a variable assignment, and that instruction scheduling has moved that assignment into the block of code for the previous statement. In this situation, the debugger would be able to consult the "range" information and warn the user that the variable had been assigned to earlier than the original source indicated.

Their "range" idea is another important part of my thesis and will be covered in a later chapter.

2.6. "Source-Level Debugging of Optimized Code: Detecting Unexpected Data Values" by Copperman

This master's thesis describes a method which had been suggested as a possibility by Zellweger to determine the currency status of variables by using reaching definitions. I did not use this portion of his work. I did find, however, that his coverage of other work in the field of debugging optimized code was excellent.

2.7. Summary

To my knowledge, all previous work has either sought to hide the effects of optimizations from the user, or else was designed to issue warnings to the user when a request with respect to the original source might not correspond to the running program. None have attempted what Zellweger briefly mentioned as a possibility: namely, the presentation to the user of a modified version of the source program which accurately reflected the underlying optimized program. Even Zellweger didn't seem to consider the possibility of showing both the original and optimized view on the screen at the same time. All current approaches with which I am familiar assume that the user will only interact with the original source program.

CHAPTER 3.
Justification Of The Present Work

3.1. Introduction

In this chapter I will present an overview of VLIW architectures and the code scheduling techniques used by VLIW compilers. I will also cover the special problems this creates for debugging and why earlier work on the debugging of optimized code is inadequate to handle these special problems.

3.2. VLIW Architecture

The most succinct definition of VLIW that I have come across is from the paper on the Multiflow architecture [3].

"VLIW computers are a fundamentally new class of machine characterized by

- A single stream of execution (one program counter, and one control unit).
- A very long instruction format, providing enough control bits to directly and independently control the action of every functional unit in every cycle.
- Large numbers of data paths and functional units, the control of which is planned at compile time.

There are no bus arbiters, queues, or other hardware synchronization mechanisms in the CPU."

3.2.1. How it is different from other architectures

VLIW processors are distinct from scalar processors in that their instruction words are actually composed of several operations, each targeted for a functional unit capable of executing in parallel with other functional units. In addition to the more conventional code generation and optimization techniques, instruction scheduling is performed in an attempt to keep as many of the functional units as busy as possible while still preserving the original source semantics.

VLIW processors are distinct from super-scalar processors (which also have multiple functional units) in that all of the scheduling of resources is handled statically. Super-scalar processors, on the other hand, manage the resources at run-time by means of extra control logic. (Of course, super-scalar processors can also benefit from static scheduling in addition to run-time scheduling.)

3.2.2. Advantages

One advantage that VLIW architectures have over super-scalar architectures is that there is no hardware devoted to synchronization issues. While this puts an extra burden on the compiler to handle all the synchronization, it also frees up real estate on the chip for more and/or faster functional units.

3.2.3. Disadvantages

One disadvantage is the flip side of an advantage listed above. Since the synchronization falls to the compiler to handle, it means that the compiler has a more complicated job than it ordinarily would. This would tend to make the performance of the program more dependent upon the compiler; if the compiler happens to contain bugs (which is more likely given the increased complexity) then the solution of turning off optimizations would be especially costly in terms of execution speed.

Another disadvantage is that there is the potential for large code size. (Larger code size means more fetches from memory and more difficulty fitting into the cache, which possibly negates some of the benefits to the VLIW architecture). This can result from some sections of code with little available parallelism in which some fields in the instruction word are NOPS (though it should be said that some architectures work around this by allowing compact versions of the instructions in which NOPS take virtually no space until actually being fetched). The added code size can also come about since some scheduling techniques result in code duplication and/or creation. Trace Scheduling, for example, relies on loop unrolling as the primary loop optimization. It also relies on the creation of new code to compensate for when an expected trace is not followed at run time.

3.2.4. The Reason for VLIW Architectures

Even in programs which show no parallelism at the source program level, there may in fact be some degree of parallelism at the machine level. For example, the statement

```
a[i] = w*x;
```

seems to involve only the calculation of the expression `w*x`, but at the level of fairly typical microcode it might require:

- An instruction fetch
- An instruction decode
- 0 or more operand fetches
- Updating the instruction pointer
- The actual multiplication
- The calculation of the address of a[i]
- The store of the result of the multiplication at that address

Depending upon the actual machine involved, many of the above steps could conceivably happen in parallel. Similarly, if fetch, decode, and execution phases can be overlapped with each other, much of the following could happen in parallel:

```
a[i] = w*x;
b[i] = x*y;
c[i] = y*z;
and so on ...
```

The main purpose behind VLIW architectures is to exploit this fine-grained parallelism that exists in ordinary code to the greatest possible extent.

3.2.5. A Brief History

In 1979, Josh Fischer published a Ph.D. dissertation concerning the optimization of horizontal microcode. In 1981, he published a paper introducing what he called Trace

Scheduling which was targeted at global microcode compaction. Finally, in 1983, he proposed a new architecture which would be specifically designed with Trace Scheduling in mind [9]. This architecture was called VLIW.

The name VLIW implies that there must be a LIW classification from which it should be distinguished. In fact, the first and last machines below would more accurately be described as LIW. However, as is the case with so much terminology, there is no clear distinction. Therefore, for the purposes of this paper, I will not distinguish between the two since the basic idea is the same: namely, that the instruction word explicitly controls each of several functional units.

Floating Point Systems introduced the FPS-164 in 1981 (before the term "VLIW" was even created) as an array processor attached to a host machine. The machine had a 64-bit instruction word able to control seven functional units at once. During a single cycle, it could perform an addition, a multiplication, two data fetches, one code fetch, the computation of the address of the next data or code word, and a branch instruction. It was designed to be programmed by hand. However, in 1984 Touzeau [23] published a paper concerning the design of a compiler for the FPS-164 computer that was able to perform many classical optimizations as well as software pipelining of many loops (those that contained only straight-line blocks of code). The paper mentions that the compiler had already been in use for 2 years.

The ELI project (ELI stands for Enormously Long Instruction) [7] at Yale was started by Josh Fischer and was meant as a project to design the compiler and hardware together to fully explore VLIW concepts. The proposed architecture had a 512-bit instruction word though no actual hardware came out of this effort.

In 1987, Multiflow Computer, Inc. announced the first commercially available machine described as "VLIW". They called it the Trace. At the same time they announced the Trace Scheduling compacting compiler [3]. This was a direct outcome from the ELI project at Yale. The Trace architecture could be configured with several different instruction widths (corresponding to different hardware configurations) from a 256-bit word (able to initiate 7 operations per clock cycle) up to a 1024-bit word (able to initiate 28 operations per clock cycle). Interestingly, Multiflow designed the machine to be a general purpose computer rather than a co-processor for scientific applications. In fact, the paper makes prominent mention that Unix was ported to the machine.

The Warp systolic array came out of a research effort at Carnegie Mellon in collaboration with GE and Honeywell. It has a 272-bit instruction word with about the same degree of parallelism as the smallest Trace configuration. The compiler for the Warp relies on software pipelining and hierarchical reduction as the primary scheduling techniques. Lam's dissertation mentions versions of the hardware in 1985, 86, and 87 [16].

The iWarp was a collaboration between Carnegie Mellon and Intel in part to produce a single chip implementation of the architecture. The architecture was outlined in 1988. This has about the same degree of parallelism as the Warp machine, but has a significantly smaller instruction word of 96 bits. It also allowed a second instruction format of 32 bits for initiating a single operation.

The Cydra 5 (from Cydrome) was introduced in 1988. It is essentially a VLIW machine with additional hardware support for overlapping loop iterations [5]. It relies on software pipelining techniques in the compiler to take advantage of the special overlapped loop support. The Cydra 5 has a 256 bit instruction word capable of controlling seven functional units.

The i860 [13] from Intel was introduced in 1989. It has two features which warrant inclusion in the list. First, it has a "dual-instruction" mode in which two instructions are executed at the same time. Second, it has some special "dual-operation" instructions which manipulate both the adder and multiplier pipelines simultaneously and which can be issued while the processor is in "dual-instruction" mode.

3.3. Instruction Scheduling

Instruction scheduling is a means by which architectures capable of overlapped execution, whether because of pipelined and/or multiple functional units, are kept as busy as possible. It is similar in many respects to the problems inherent in operating system design; namely, it attempts to find the most efficient utilization of the resources at hand.

Instruction scheduling for scalar machines involves only instruction re-ordering. (Recall that the scheduling for the machine mentioned in the DOC paper [4] was intended to fill branch-delay slots and to avoid processor stalls on memory accesses.) Instruction scheduling for VLIW machines, on the other hand, is much more complicated. A compiler for a VLIW machine will not only re-order operations; it will also pack several operations into single instructions since the hardware contains more than one functional unit capable of executing in parallel. For example, depending upon the source code and available hardware resources, the instruction scheduler may create single machine instructions which contain code corresponding to portions of several source lines and different loop iterations.

3.3.1. Compaction and Re-ordering

Code compaction and re-ordering is the simplest kind of instruction scheduling for VLIW architectures. It does not involve the duplication or creation of additional code as the following techniques do. It effectively takes a list of operations comprising a basic block and arranges them subject to two constraints: first, the original program semantics must be preserved; second, the machine resources must be kept as busy as possible.

3.3.2. Trace Scheduling

Conditional branches ordinarily cause problems because processors cannot "know" ahead of time which path will be taken, and therefore cannot continue fetching and executing until the path is known. Many processors get around this by rearranging code to put an instruction in the branch-delay slot which is safe to execute no matter which branch is taken. This is limited, though, by the fact that it may not be possible to find a suitable instruction. Also, since the branch still exists, the scheduling of resources cannot continue until the new branch is started. Trace Scheduling takes a different approach: based on some heuristic, it chooses the most likely path (up to the next backwards branch, if possible), treats the entire path as one long basic block, and packs operations along this path into instruction words. The compiler also inserts compensation code into the off-trace branches to compensate for when the expected trace is not taken at run-time. Since branches are effectively removed yielding much larger basic blocks, the fine-grained parallelism can be exploited over much greater extent [7]. In other words, because of the compensation code, the scheduler is free to move operations beyond the boundary of the basic block.

Trace scheduling is based on the assumption that there is a much higher probability of a branch going one direction rather than the other. Fortunately, this is often the case. The assumption is certainly true for loops, in which the branch back to the beginning of the loop is taken every time except for the last iteration. It is also true for error-checking code in which the error condition rarely occurs. On the other hand, programs containing mostly branches that go either direction with about equal probability are not going to see much, if any, benefit from the Trace Scheduling technique.

One of the biggest disadvantages to Trace Scheduling is that if the program execution does not follow the trace that was originally predicted -- that is, if the heuristic is poorly chosen -- then the compacted code may perform poorly [6]. To guard against this, the programmer must add another stage to the program development cycle. The program must be run with realistic input to check the execution path taken at run-time. This data can then be fed to the compiler to fine-tune its branch predictions.

Another disadvantage is that there can be significant code expansion. One reason for this is because of the compensation code that is necessary to preserve program correctness. It can also be a problem because the primary technique used on loops is unrolling followed by scheduling. This effectively duplicates the source lines of loop bodies in order to achieve longer loop bodies, and hence, longer basic blocks.

3.3.3. Software pipelining

Software pipelining is an instruction scheduling technique that is applied only to loops. It is based on the observation that a "next" iteration of a loop may start before previous ones have finished if all of the needed data from previous iterations are available, and if it does not require resources that are already in use. The goal is to find the best combination of the schedule for a single iteration, and the initiation interval for starting successive iterations.

There are ordinarily 3 distinct portions of a software pipelined loop. First, there is the prolog in which more and more iterations start. This is essentially for the purpose of "filling" the software pipeline. Second, there is the steady-state in which iterations start and finish on a regular schedule. The loop will spend most of the time in this portion. Third, there is the epilog in which the pipeline is "drained".

One advantage to software pipelining is that it results in relatively compact code size. Another is that it frequently results in the optimal schedule [16]. There are at least two disadvantages: it may only be suitable for loops having a small body and may not be all that suitable for nested loops.

3.3.4. Hierarchical Reduction

Ordinarily, a simple *if* statement could prevent a loop from being software pipelined. This was certainly true for software pipelining as implemented in the compiler for the FPS-164 [23]. Hierarchical Reduction is a scheduling technique implemented by

Lam as a companion to software pipelining [16]. The motivation of it was to enable loops with conditional constructs to be software pipelined. The basic idea is to schedule the program from the innermost constructs out, replacing the construct with a single node which is then scheduled with its surroundings. This process is continued until the entire program is reduced to a single node.

3.4. Why Scheduling for VLIW Architectures is a Problem for Debugging

Debugging non-optimized code is simple because it has the following characteristics: the code is generated (and executes) statement-by-statement; and variables generally live at only one location during their entire existence. Debugging optimized code can be a big problem because the code no longer has those characteristics. Debugging VLIW code after instruction scheduling is an even larger problem. First, there is the instruction re-ordering. At the very least, this causes the code from different statements to be interleaved in some fashion. In this regard, VLIW code is no worse than typical optimized code (which is bad enough). However, in addition to instruction re-ordering, there is typically a compaction phase in which several operations are packed together into a single instruction word. Thus a single instruction word can contain code from portions of several source lines. Finally, there are also some transformations performed on the code in order to achieve higher performance. To take one example, software pipelining can transform the code in such a way that individual instructions can contain code from different loop iterations. That is, there can be several copies of a particular source line in various stages of execution at the same time. If there can be several copies of a line in execution at the same time, that means that there can be several copies of a variable in existence at the same time, each with a potentially different yet valid value.

3.5. Why we want to be able to debug this code

All of the reasons for wanting to debug optimized code are valid for VLIW code after instruction scheduling. Among those reasons are:

- The performance of these kinds of machines is their main reason for existence. Therefore, if a math library, for example, exhibits some bugs after instruction scheduling, it is not feasible to ship unoptimized and/or unscheduled code since the performance may be orders of magnitude slower.

- The programmer should be able to debug the same code that will be in the final product.

- Since the code generation for VLIW machines is more complicated than for conventional machines, the likelihood of compiler bugs is increased. Compiler writers would like to be able to debug at the source-level; and programmers who encounter compiler bugs would like to be able to locate the problem and work around it.

- Debugging at the assembly language level is tedious for conventional machines; for VLIW machines it is even worse.

CHAPTER 4.
An Approach To Debugging VLIW Code After Instruction Scheduling

4.1. Introduction

I will show that effective debugging of VLIW code after instruction scheduling can be achieved by a combination of:

- enhancements to the conventional code and data debug information maps to allow the user to make expression-oriented (rather then line- or statement-oriented) queries.
- a highlighting scheme (plus a modified source when needed) to show the current state of the computation. In this way, the user is informed graphically what portions of the program have executed.
- a re-definition of some of the traditional debugger commands, since the notion of the "current" location can become fuzzy in the presence of instruction scheduling.

4.2. Scope Of This Proposal

I am targeting this proposal at instruction scheduling techniques which involve the re-ordering of operations, combined with the compaction of operations into long instruction words, combined with code duplication transformations such as loop unrolling or software pipelining. I should also state that this proposal is not intended for other optimizations, though I will consider them informally later.

Some instruction scheduling techniques can result in the creation of code that corresponds to nothing in the original source. It is my understanding that this is possible with the Trace Scheduling technique.

For example, suppose that as a Trace Scheduling compiler proceeds, it has picked a "most likely" trace and has begun to re-arrange and compact the code along the trace. If an operation is moved down the trace to a position past a conditional branch to an off-trace target (called a split), or up the trace to a position prior to where an off-trace branch joins the main trace (called a join), then the compensation code would consist of simply copying that same operation to the off-trace branch so that it is executed no matter what branch is taken at run time. Such code should not create any significant problems for debugging beyond the fact that the debugger must deal with two object code sequences mapping to the same source expression.

However, if the compiler moves an operation up the trace to a position before a conditional branch, the compensation code is potentially much more complicated. It would have to somehow undo the operation in the event that the off-trace branch is taken at run-time. The problem this creates for debugging is that between the time that the operation is completed (which should not have happened) and the time that it is un-done (to restore the semantics of the program), the program is not semantically equivalent to the original. I am not proposing any solution for such techniques.

4.3. Enhancements To Conventional Debug Information

In this section, I want to demonstrate that effective debugging of VLIW code after instruction scheduling requires code and data maps that are sufficiently detailed that they are able to handle queries at the expression level as opposed to line or statement level. To do that, I will work through a progression of scenarios showing increasingly complicated instruction scheduling, each of which requires more detail in the debug information maps.

4.3.1. Conventional Debug Information

The code map for non-optimized code has typically consisted of a mapping from statement numbers to blocks of corresponding object code. This is largely sufficient for such code because the code is typically generated on a source-statement by source-statement basis; that is, there really *is*, for the most part, a contiguous block of object code for each source statement.

Consider the following source code fragment (numbers in the left-hand column are line numbers):

```
1:    int i1, i2, i3;
2:    float f1, f2, f3;
3:
4:    i1 = i2 + i3;
5:    f1 = (f2 / f3) / i1;
6:    return;
```

And now some possible assembly language code (numbers in the left-hand column are offsets from the beginning of the code):

```
0:    load r1,  i2
1:    load r2,  i3
2:    add  r1,  r2
3:    store     r1,  i1
4:    load fr1, f2
5:    load fr2, f3
6:    fdiv fr1, fr2
7:    load fr2, i1
8:    fdiv fr1, fr2
9:    store     fr1, f1
10:   ret
```

The conventional code map might appear as follows:

Line Number	Instruction Offset
4	0
5	4
6	10

It would be interpreted to mean that the object code corresponding to line 5, for example, starts at the instruction at offset 4 and continues through the end of the instruction at offset 9.

The conventional data map might appear as follows:

Name	Type	Location
i1	integer	100h
i2	integer	104h
i3	integer	108h
f1	real	10ch
f2	real	110h
f3	real	114h

It would be interpreted to mean that the given variable is of the listed type and can be found at the listed location on statement boundaries.

Here is an example of how they might be used during a debugging session. Suppose that the user wants to set a breakpoint at line 5 (typically, this is interpreted to mean "break such that line 5 will be executed next"). In this event, the debugger consults the code map and finds that the instruction at offset 4 is the beginning of the code for line 5. The debugger then saves the original instruction, replaces it with a special breakpoint opcode, and then starts execution. When the breakpoint is hit, the debugger returns the original opcode to its location, and then updates the screen giving some indication that line 5 is next. This "indication" typically consists of highlighting the source line in some manner. At this point, the debugger will wait for user input. This input might consist of queries for variable values, for example. In this case, the debugger consults the data map, finds the corresponding memory location, retrieves the value, and then prints it according to its type. In the event that the user wishes to modify a variable, the debugger finds the location and places the desired value there.

Suppose that during a debugging session a division-by-zero exception occurs at the instruction at offset 6. The debugger consults the code map and discovers that the instruction at offset 6 in the object code belongs to the block of code corresponding to line 5 in the source, and reports that as the source line responsible for the error. In order to determine the cause of the problem, the user can query the debugger for the values of any program variable. To carry this out, the debugger consults the data map, retrieves whatever values the user wants and then prints them.

Even in this simple scenario, the conventional debug maps may not be perfect. This can happen on those occasions when the user gets control at some point after a statement has begun executing, but before it has completed. For example, assume that a source statement consists of several operations. If an exception occurs after some operations have been executed but before memory locations have been updated; and if the user then queries the debugger for a variable in one of those locations, the debugger would retrieve and print the value of the variable on the previous statement boundary. However, for the most part, debugging of non-optimized code is very straightforward with these simple maps.

4.3.2. Re-ordered Code

Now assume that the target architecture has independent integer and floating point units, and that instruction scheduling has been performed. Here is one possible result (refer to the previous source fragment):

```
0:   load     fr1, f2
1:   load     r1,  i2
2:   load     fr2, f3
3:   load     r2,  i3
4:   fdiv     fr1, fr2

5:   add      r1,  r2
6:   store    r1,  i1

7:   load     fr2, i1
8:   fdiv     fr1, fr2
9:   store    fr1, f1
10:  ret
```

The code no longer executes on a statement-by-statement basis; therefore, the debug information needs to be modified in order to avoid being incorrect.

One possible solution is described in the DOC paper [4]. They associated a statement marker with the first instruction for each source statement. In the event that an optimization phase moves such an instruction, the marker would be moved to the next instruction in the block. For the above code, this would have the effect of having the statement marker for line 4 being associated with instruction 5, and the statement marker for line 5 being associated with instruction 7. The instructions that come before the line 4 marker would be treated as if it belongs to the block of code for the previous line. It is not perfect, but it is useful since it largely maintains the illusion that the code is executing on a statement-by-statement basis. Here is a code map which incorporates such a modification.

Line Number	Instruction Offset
4	5
5	7
6	10

Here is what the data map might look like. It is essentially an example of what was described in the DOC paper [4] which shows the location of the variable as a function of the instruction pointer. This modification is necessary because the re-ordering of the code (as well as other optimizations) can take away an essential characteristic of non-optimized code: that the memory locations of all variables will be up-to-date on all statement boundaries.

Name	Type	Location	IP Range
i1	integer	r1	6
		fr2	8
i2	integer	r1	2-5
i3	integer	r2	4-5
f1	real	fr1	9
f2	real	fr1	1-4
f3	real	fr2	3-4

There is still one more part to any solution for this kind of code. Typical debugger commands need to be re-considered. For example, what does it mean to tell the debugger to "break at line 5"? In non-optimized code it is unambiguous. However, when instructions have been re-ordered across statement boundaries it can be interpreted two ways. It could mean to break before any instruction that belongs to line 5 (the instruction at offset 0) or it could mean to break after the last instruction of the line before line 5 (the instruction at offset 7). Note that for non-optimized, non-re-ordered code, the two will be the same. Zellweger considered this problem, and introduced terminology that is used to distinguish them. A breakpoint placed before the first instruction of the desired statement is called a "semantic" breakpoint; one placed after the last instruction of the previous statement is called a "syntactic" breakpoint. A debugger for this kind of code must either implement both kinds of breakpoints, or treat all breakpoint requests as being all of one kind. (The DOC debugger in effect implemented all breakpoint requests as syntactic breakpoints.) The solution is not bad since the debug information is not much more complicated than for non-optimized and yet is useful. However, it is not perfect. The unsuspecting user might be very confused to discover that setting a breakpoint on line 4 somehow resulted in a portion of line 5 being executed.

4.3.3. Code Map for Re-ordered and compacted code

Now assume a VLIW architecture with multiple integer and floating point units. In this scenario, the above described debug information is again insufficient. In order to see why, assume that we start with the above example of re-ordered code, and then pack several operations into individual VLIW instructions. In this contrived example, I assume that the VLIW word has four slots: the first two for the integer units and branch-type instructions, and the second two for floating point units. Here is one possible result (the number in the left column is an offset from the beginning of the code):

```
0:  load r1,i2    load r2,i3  load fr1,f2   load fr2,f3
1:  add r1,r2     nop         fdiv fr1,fr2  nop
2:  store r1,i1   nop         nop           nop
3:  nop           nop         load fr2,i1   nop
4:  nop           nop         fdiv fr1,fr2  nop
5:  nop           nop         store fr1,f1  nop
6:  ret
```

The code map in the previous section is insufficient because individual VLIW instructions may contain code from portions of several source lines. In a sense, this is the inverse of conventional code in which source statements map to several machine instructions. Therefore, it makes sense to map instructions to a list of corresponding source expressions. In an actual implementation, the corresponding source might more simply be represented as a simple offset from the beginning of the source file.

Here is one possible result:

Instruction Offset	Corresponding Source Expression
0	4:load of i2; 4:load of i3; 5:load of f2; 5:load of f3
1	4:add; 5:division of f2 by f3
2	4:assign
3	5:load of i1
4	5:division by i1
5	5:assign
6	6:return

Assuming a division-by-zero exception during the execution of the instruction at offset 1, the above map would be used by the debugger to highlight two expressions rather than two entire source lines. Any more detail would have to come from the architecture. That is, if the architecture can make available to the exception handler which portion of which instruction caused the exception, then the debugger could certainly pass the information along to the user.

No additional changes need to be made to the data map for this level of instruction scheduling. This is because the previously presented data map can already handle cases where variables can be found in registers on statement boundaries.

4.3.4. Code Map for Duplicated, Re-ordered, and Compacted Code

Finally, consider the complication that arises when a scheduling technique which uses code-duplication has been performed. Software pipelining and loop-unrolling followed by code compaction are two examples. The following is a slightly more complex example derived from the Microarch paper [15]. It was chosen because it lends itself to software pipelining. Notice that each line in the body of the loop (except for the first line) has a data dependence on the previous line, and that each loop iteration (because of the first line in the loop body) has a dependence on the previous iteration. Most other examples I have used are in C, but this one is in Fortran.

```
subroutine micro2(a,b,c,d,n)
real a(n),b(n),c(n),d(n)
integer i
do 10 i = 1, n
      a(i+1) = a(i) + 1
      b(i) = a(i+1) / 2
      c(i) = b(i) + 3
      d(i) = c(i)
10    continue
end
```

The following is a possible schedule for one iteration of the loop body. I have taken some liberties with the pseudo-assembly language code in an attempt to keep the example clear. For example, I am allowing array references in the assembly language instructions. (Again, the number in the left-hand column is an instruction offset from the beginning of the code.)

```
0     load r1, a(i)
1     inc r1
2     store r1, a(i+1)

3     load r2, a(i+1)
4     div r2, 2
5     store r2, b(i)

6     load r3, b(i)
7     add r3, 3
8     store r3, c(i)

9     load r4, c(i)
10    store r4(i)
```

After software pipelining, the following pseudo-assembly code is obtained. As in the previous example, I am taking quite a few liberties with the code in an attempt to keep it clear:

- it contains array references.
- a real compiler would presumably perform many additional optimizations on the code. One example is that no attempt has been made to take advantage of values which may already be available in registers.
- I assume that loop control registers are initialized upon entry. Thus, the machine "knows" that the body of the loop runs from offset 9 through 11.
- the notation exemplified by `"1..n-3"` in many instructions in the body of the loop is meant to indicate that the instruction will execute for all values of `i` between `1` and `n-3`.

Overlapped iterations begin at the instruction at offset 3 during which iteration 1 continues and iteration 2 starts. Overlap increases until the steady state of the loop is reached at offset 9. While the code continues in the steady state, there are four iterations of the loop in various stages of execution at once. (Again, the number in the left-hand column is an instruction offset from the beginning of the code.):

```
0    load r1, a(1)       nop                 nop                 nop
1    inc r1              nop                 nop                 nop
2    store r1, a(2)      nop                 nop                 nop

3    load r2, a(2)       load r1, a(2)       nop                 nop
4    div r2, 2           inc r1              nop                 nop
5    store r2, b(1)      store r1, a(3)      nop                 nop

6    load r3, b(1)       load r2, a(3)       load r1, a(3)       nop
7    add r3, 3           div r2, 2           inc r1              nop
8    store r3, c(1)      store r2, b(2)      store r1, a(4)      nop

9    load r4, c(1..n-3)  load r3, b(2..n-2)  load r2, a(4..n)    load r1, a(4..n)
10   store r4, d(1..n-3) add r3, 3           div r2, 2           inc r1
11   nop                 store r3, c(2..n-2) store r2, b(3..n-1) store r1, a(5..n+1)

12   nop                 load r4, c(n-2)     load r3, b(n-1)     load r2, a(n+1)
13   nop                 store r4, d(n-2)    add r3, 3           div r2, 2
14   nop                 nop                 store r3, c(n-1)    store r2, b(n)

15   nop                 nop                 load r4, c(n-1)     load r3, b(n)
16   nop                 nop                 store r4, d(n-1)    add r3, 3
17   nop                 nop                 nop                 store r3, c(n)

18   nop                 nop                 nop                 load r4, c(n)
19   nop                 nop                 nop                 store r4, d(n)
20   nop                 nop                 nop                 nop
```

The modification I propose to the code map to handle this situation is for the compiler to indicate in the debug information that multiple copies of a source line can be in execution at once. This can be achieved by an additional field in which the compiler could indicate the relative iteration in addition to the source expression for each machine instruction. This information would then be used by the debugger to create and present to the user the state of the program.

Here is one way the code map could be constructed:

Instruction offset	Corresponding source	Iteration	Comments
0	line 5 read	1	overlap: 1; loop prolog
1	line 5 add	1	
2	line 5 assign	1	
3	line 6 read	1	overlap: 2
	line 5 read	2	
4	line 6 div	1	
	line 5 add	2	
5	line 6 assign	1	
	line 5 assign	2	
6	line 7 read	1	overlap: 3
	line 6 read	2	
	line 5 read	3	
7	line 7 add	1	
	line 6 div	2	
	line 5 add	3	
8	line 7 assign	1	
	line 6 assign	2	
	line 5 assign	3	
9	line 8 read	1	overlap: 4; loop body
	line 7 read	2	
	line 6 read	3	
	line 5 read	4	
10	line 8 assign	1	
	line 7 add	2	
	line 6 div	3	
	line 5 add	4	
11	line 7 assign	2	
	line 6 assign	3	
	line 5 assign	4	
12	line 8 read	2	overlap: 3; loop epilog
	line 7 read	3	
	line 6 read	4	
13	line 8 assign	2	
	line 7 add	3	
	line 6 div	4	
And so on ...			

Finally, we have a code map which is able to deal with much of the complexity caused by VLIW architectures and instruction scheduling: namely, that the code may have been rearranged, compacted, and duplicated.

4.3.5. Data Map for Duplicated, Re-ordered, and Compacted Code

The DOC paper described a very useful approach to tracking a variable's location as it changed during the course of a program's execution. Unfortunately, though, the solution did not extend to array references [4]. In order to be useful in the event of aggressive code scheduling, however, this needs to be implemented since so much of instruction scheduling involves array references within loops.

Recall that in the case of code duplication, several iterations of a loop can be in various stages of execution at once. If the user wants to know the value of a(i) when "i" is equal to 2, for example, the debugger must somehow be able to determine that array elements corresponding to different values of "i" may live in different places.

One possible solution is a straightforward extension to the original idea presented by Coutant, et. al. [4] such that references to individual array elements corresponding to different values of the loop iteration variable would result in separate entries in the data map as if they were scalar references. The data map is on the following page.

Name	Type	Location	IP range
i	integer	a function of one or more loop control registers	all
a(i), i = 1	real	register r1	1
a(i), i = 2	real	register r1	4
a(i), i = 3	real	register r1	7
a(i), i > 3	real	register r1	10
a(i+1), i = 1	real	register r1	2-3
		register r2	4
a(i+1), i = 2	real	register r1	5-6
		register r2	7
a(i+1), i = 3	real	register r1	8
a(i+1), i > 2 and i < n	real	register r1	11
		register r2	10
a(i+1), i = n	real	register r2	13
b(i), i = 1	real	register r2	5-6
		register r3	7
b(i), i = 2	real	register r2	8
b(i), i > 1 and i < n-1	real	register r2	11
	real	register r3	10
And so on ...			

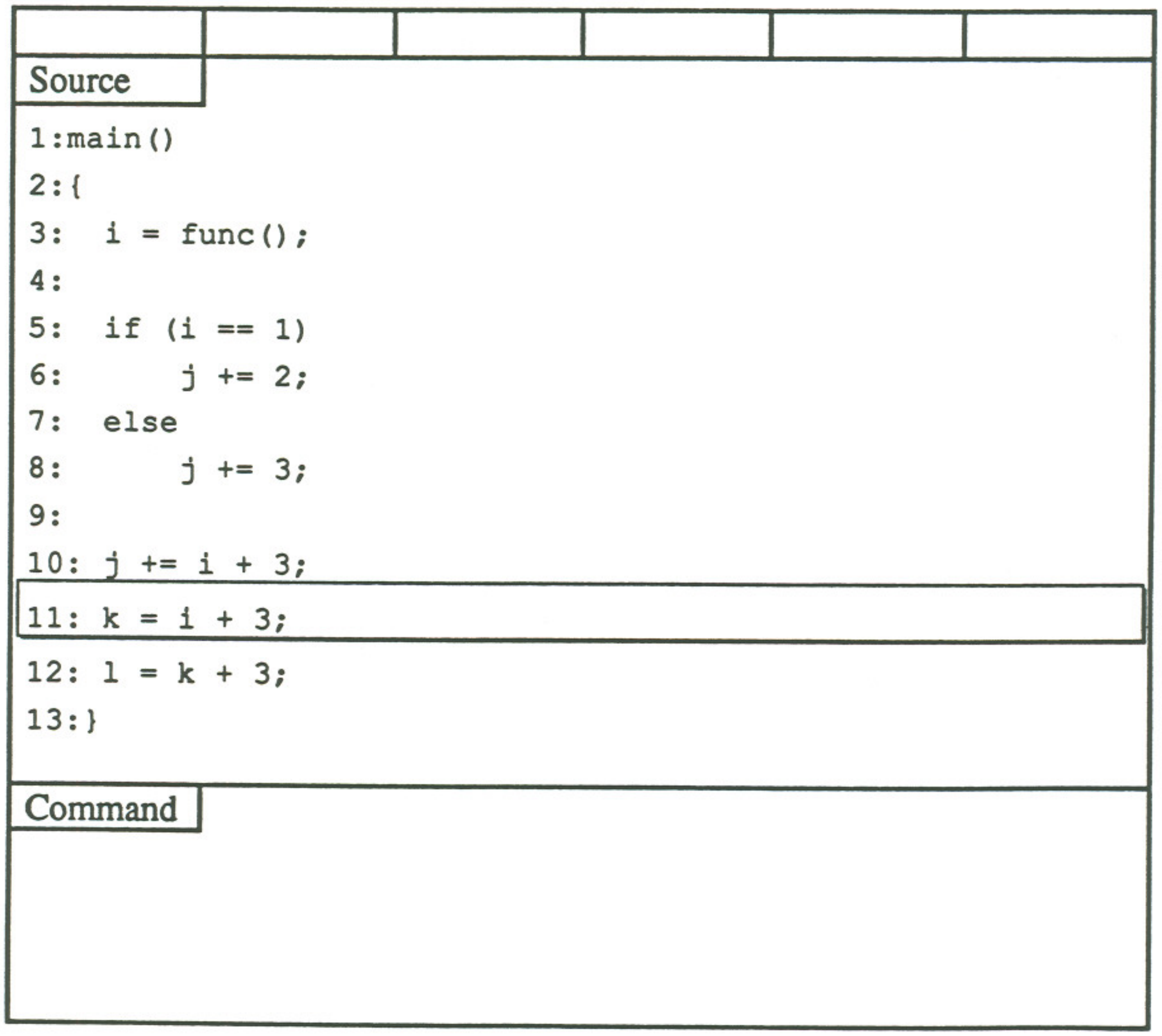
4.4. Representing The State Of The Computation

Interactive source-level debuggers for non-optimized code with which I am familiar have a similar look about them. They typically display several windows at once with a variety of information; however, the dominant feature is usually a window containing a copy of the user's source code with one particular source line highlighted. The highlighting is meant to indicate the source line corresponding to the current instruction pointer. In other words, the highlighted statement will ordinarily be the next one to execute.

Setting a breakpoint is usually handled in one of two ways: the user might type in the command on the command line (e.g., break at line 5); the other method would be to move the cursor to the desired location and hitting a dedicated mouse or keyboard button.

Querying for the value of a particular variable is handled similarly: either by means of an explicit command or by pointing the cursor at it and hitting a button.

Figure 4.1 might represent what the user would see during the debugging of a non-optimized program:



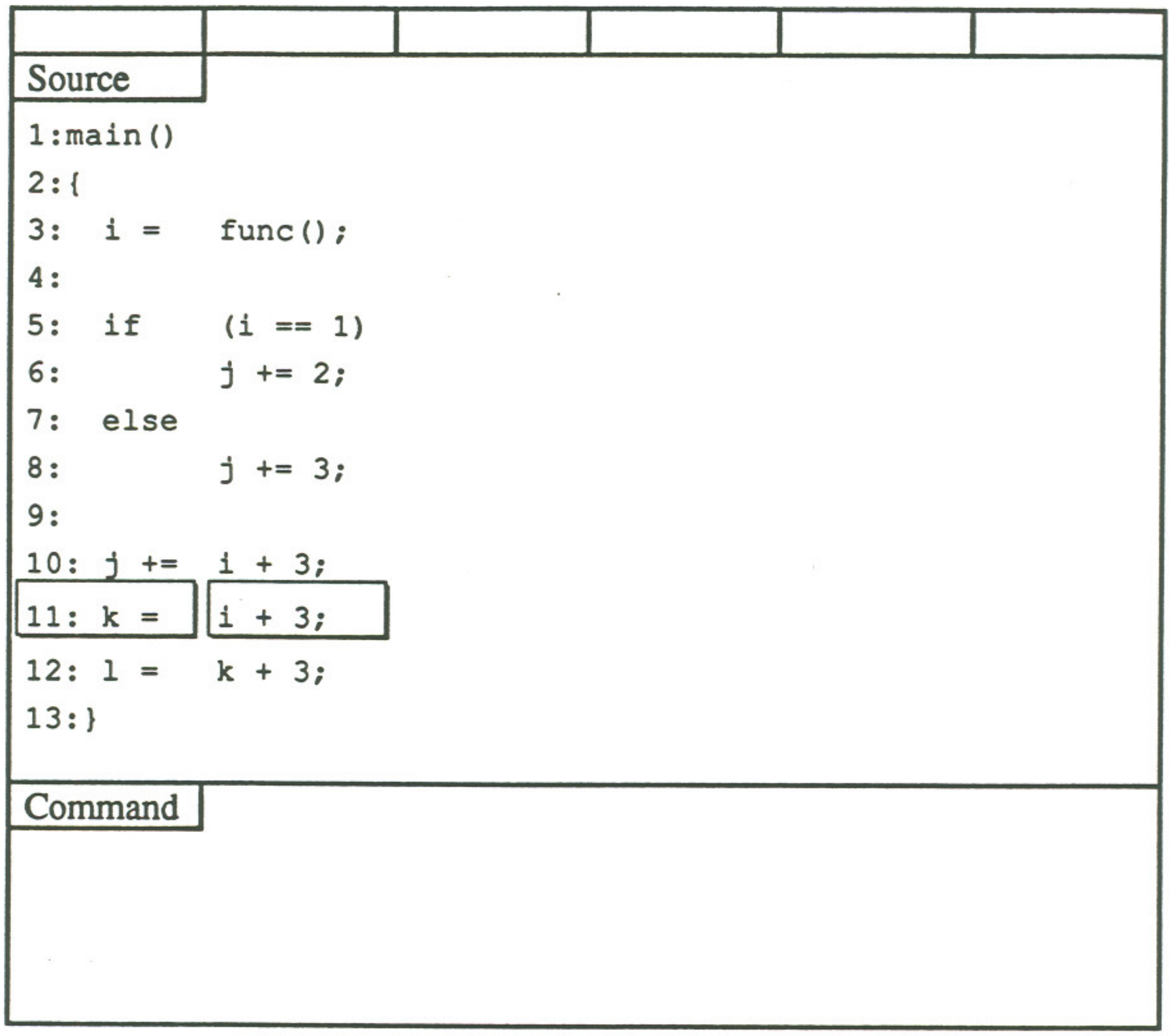
```
1:main()
2:{
3:   i = func();
4:
5:   if (i == 1)
6:        j += 2;
7:   else
8:        j += 3;
9:
10: j += i + 3;
11: k = i + 3;
12: l = k + 3;
13:}
```

Figure 4.1. Source Level Debugger Screen for Non-optimized Code

The user would immediately know that the highlighted statement corresponds to the current instruction pointer. Due to the nature of non-optimized code, the user would also be able to determine to a great extent the overall state of the computation; i.e., which other statements have executed and which have not. This in turn gives the user the necessary information to make intelligent queries about the values of program variables.

I had to qualify the previous statements somewhat because of the existence of conditional branches. If the current location, i.e., the highlighted statement, is somewhere past the end of an if-else or some other kind of branching statement, then it might be difficult or impossible to determine which branch was taken when it was encountered. "Difficult" means that the user must determine the value of the variable used at the place where the condition was tested, and infer from that which branch was taken. "Impossible" means that the value used in the conditional test is no longer available. (It may be that the storage for that variable is in use by another variable, or it may be that the value returned by a function was used directly in the conditional test). To sum up: when the user knows the current location in a non-optimized program, the user also knows a lot about the overall state of the computation.

Contrast that with what can be assumed about the state of a VLIW program. Figure 4.2 represents what the user might see during a debugging session for VLIW code which has undergone instruction scheduling. The debugger has highlighted the expressions corresponding to the current VLIW instruction:



```
1:main()
2:{
3:   i =    func();
4:
5:   if     (i == 1)
6:          j += 2;
7:   else
8:          j += 3;
9:
10: j +=   i + 3;
11: k =    i + 3;
12: l =    k + 3;
13:}
```

Figure 4.2. Source Level Debugger Screen for VLIW Code

As in the previous example, the user would immediately know that the highlighted expressions correspond to the current value of the instruction pointer. Unlike the previous example, however, the user cannot make any easy guesses about which other statements have executed and which have not (that is, unless the user can do data and control flow analysis on the fly). To sum up: when the user knows the current location in a VLIW program which has undergone instruction scheduling, that's about all that can be known about it. The rest of my thesis is a proposal to remedy that situation.

Previous work in the field of debugging optimized code attempted to convey to the user the state of the computation by two approaches:

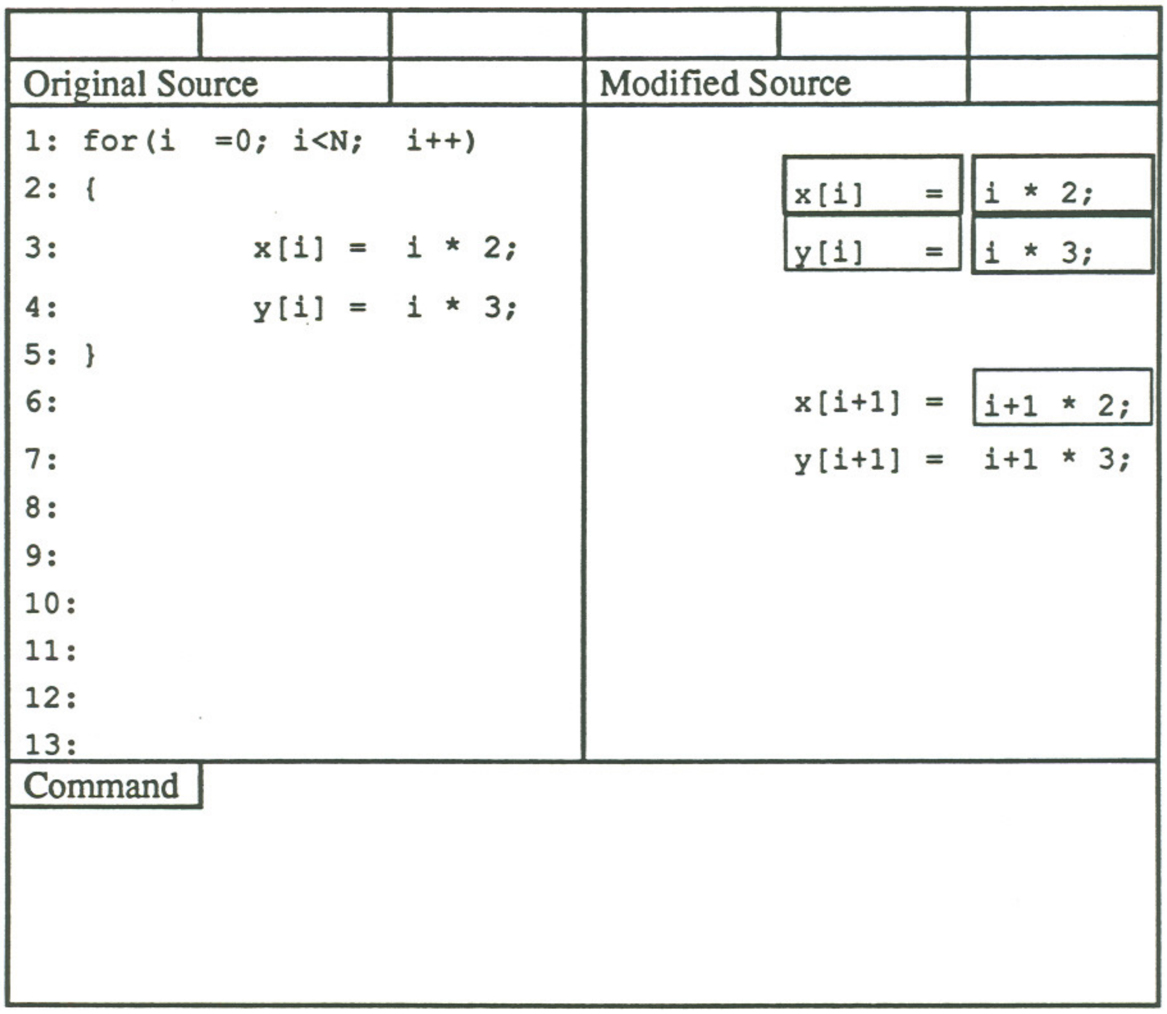
- When possible, maintain the illusion that the code is not optimized, and in so doing, allow the user to infer from the current location the overall state of the computation. (Zellweger managed to achieve this for procedure in-lining, even going so far as to have the in-lined procedure appear on the call stack [30].)
- When not possible, inform the user by some appropriate means how the actual state of the program differs from the apparent (i.e., non-optimized) state. In the literature I have seen, this means printing out a message to explain the situation. (This is typified in the DOC debugger which allowed the re-ordering of instructions across statement boundaries. If the instruction which updated a variable was moved into the block of code for a previous statement, and if the user requested a breakpoint on the original statement, the debugger might issue a message to the effect that the variable had had an "early" update [4].)

I propose a different approach.

First, when necessary, the debugger will use information described earlier in this chapter and provided by the compiler to create a modified version of the source and display that along side the original as an "explanation" of whatever transformed the source. This actually was suggested as a possibility by Zellweger; however, she appeared to be against it as it might overwhelm the user. For purposes of this thesis, I am limiting my consideration to software pipelining and loop unrolling as examples of when this might be necessary. I believe it unlikely that these would overwhelm anyone.

Second, I propose that this possibly modified source be highlighted appropriately as the program runs to indicate what portions of the program have been executed. (I would like to take credit for this idea but it was actually suggested to me in the course of looking for a thesis topic. [18]) In this way, the user will have the same kind of information about the current state of the running program as the person debugging non-optimized code.

Figure 4.3 represents one possible implementation. The code in this example has undergone loop unrolling followed by instruction scheduling. The unrolled loop body is shown on the right half of the screen. The dark highlighting indicates that the corresponding expression has completed; the light highlighting indicates that the corresponding expression is next; no highlighting indicates that the expression has not been touched.



```
Original Source                        Modified Source
1: for(i  =0; i<N;  i++)
2: {                                                        x[i]     =   i * 2;
3:             x[i] =  i * 2;                               y[i]     =   i * 3;
4:             y[i] =  i * 3;
5: }
6:                                                          x[i+1]  =   i+1 * 2;
7:                                                          y[i+1]  =   i+1 * 3;
8:
9:
10:
11:
12:
13:
Command
```

Figure 4.3. Improved Source Level Debugger Screen for VLIW Code

This poses a big problem for the debugger, however. Ordinarily, the debugger will only gain control at breakpoints and program exceptions. How, then, is it supposed to determine which statements have executed? To phrase it another way: before the debugger can indicate to the user the overall state of the program, the debugger must first *determine* the state of the computation. I propose that that raw data, consisting of which portions of a program have executed and which have not, can be obtained while the program is running by *path determination.*

Recall that for Zellweger, path determination was only a means to an end. Her main goal was not to show the user what path was taken through the code; rather, her goal was to unravel the effects of the tail-merging optimization on debugging. Determining the actual path taken through a tail-merged region of code was her means of achieving it.

My proposal is to use the same basic technique as Zellweger, i.e., path determining breakpoints and time stamps, but I would put them to a different use. My intent would be to determine what path was taken through the code and then to pass that information along to the user by means of highlighting the (possibly modified) source appropriately.

I can think of several ways this could be accomplished. Here is one way that requires nothing additional of the compiler except what has been covered already. The extra work would be handled during the debugging session by the debugger:

- During debugging, the debugger maintains a global variable which is initially zero, and which is incremented before each use. This is used as a time stamp as discussed below.

- The code map as received from the compiler is used to create a data structure which will be used to collect path determiner information at run-time. It will have two fields in addition to the ones provided by the compiler. One is for holding the opcode of a path determining instruction. The other is for holding the value of the time stamp.

- The debugger scans the entire program image in memory and builds a control-flow graph representation of the basic blocks.

- Path determining instructions are located using the control-flow graph. These are all of the predecessors of target instructions which have more than one predecessor (thus the need to determine through which predecessor control passes at run time).

- Path determining instructions are copied to their slot in the code map.

- Breakpoint instructions are copied to all path determining locations.

- All time stamp entries are initialized to zero.

Now the debugger is ready for the debugging session to begin. Here is the sequence of events in a rough pseudo-code:

```
Repeat until the program exits or the user gets control

{

    Execute until the program exits or the next exception occurs.

    The debugger gets control due to an exception.

    If the cause is something other than the execution of the breakpoint opcode,

        Update the screen as described below, and hand control to the user.

    Else (The exception is due to execution of a breakpoint instruction)

        The debugger searches the code map for the address of the instruction.

        The debugger checks to see what kind of breakpoint it is.

        If it is something other than a path-determining breakpoint,

            Update the screen as described below, and hand control to the user.

        Else (It must have been a path-determining breakpoint)

            Increment the time stamp and copy the new value to the code map.

            Copy the real instruction to its location.

            Single step it.

            Put the breakpoint opcode back in.

        Endif

    Endif

}
```

When the user gets control, it is time to update the screen. Here is how it could be accomplished. It is something of a brute-force approach for the sake of simplicity.

Use the code map, the data map, and the control-flow graph to:

- find all expressions in the source corresponding to the current instruction and highlight them appropriately.
- Work backwards from the current instruction until an instruction with multiple predecessors is encountered, highlighting all corresponding source expressions along the way.
- Follow the path through the predecessor with the highest time stamp value. In the event that this predecessor is actually the branch backwards at the end of a loop, follow this path only if the body of the loop has not been touched yet.
- Continue as far as needed.
- Find all variables which are in expressions which have been executed and whose range does not include the current instruction pointer and highlight them in some other fashion. This will give a visible indication to the user that the variable is not available.
- Optionally highlight all other expressions corresponding to instructions which have a time stamp greater than zero. This should catch all expressions which have executed at some point: for example, during previous iterations of a loop.

The processing of the time stamps in the vicinity of a loop perhaps needs a little more explanation. If it is not a software-pipelined loop, then the result would be that only those expressions which executed during the last iteration would be highlighted. This is just what is needed.

What about the case of a software-pipelined loop, for which there will be a copy of the source lines on the screen for each iteration that can be overlapped? This is really not much more complicated. If the starting place (for working backwards) is the loop prolog, then the value of the loop counter would indicate which copy of the lines in the modified source to highlight. If the starting place is in the body of the loop, then the loop is in its "steady state". This means (for our purposes) that it doesn't matter if this is the first time into the body of the loop or the hundredth time around the loop: the loop has reached a steady state. Therefore, it can be treated as if it were the first time into the loop which means that we can just work backwards through the loop entry, and backwards through the loop prolog highlighting as appropriate.

A more efficient implementation would take into account the number of lines of source which can be displayed at once. This would be used to find the corresponding range of instructions which would give a limit to how far back in the control flow graph to go.

At the end of this process, every expression on the screen will be highlighted in a fashion to indicate one of several possible states:

- Never executed
- Executed at some point in the past
- Executed during the most recent iteration of a loop or the most recent time through a conditional branch

Furthermore, every variable on the screen will have highlighting to indicate one of several possible states:

- Never touched
- In an expression which has executed, and the value can be determined by the debugger
- In an expression which has executed, but the value cannot be determined by the debugger

To illustrate these ideas, consider figure 4.4. To simplify the example, the code map will map the source code directly to the control-flow graph. Also, there are no array references, code-duplication or even scheduling involved.

```
1:     if (function() == TRUE)
2:         i += 1;
       else
3:         i += 2;
       endif
4:     i++;
5:     while (i < 10)
6:         i++;
7:     printf("i = %d\n", i);
8:     And so on...
```

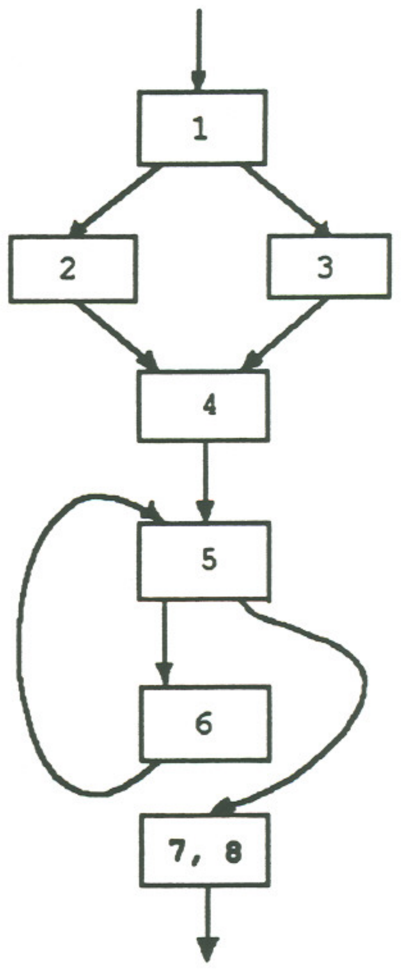


Figure 4.4. Source Fragment and Control-Flow Graph

By examining the control-flow graph, you can see that blocks 4 and 5 have multiple predecessors. Therefore, path determining breakpoints would be placed at the end of blocks 2, 3, 4, and 6. The time stamp entries for these path determining breakpoints are all initialized to zero. The other entries can be initialized to some special value indicating that time stamps are not applicable. The user has placed a breakpoint at line 8.

Here is the simplified code map as it might appear just prior to execution:

Source Line	Time Stamp
1	none
2	0
3	0
4	0
5	none
6	0
7	none
8	none

During execution, either the path determining breakpoint at block 2 or 3 will be hit a single time; the one at block six will be hit 0 or more times. These will all be processed as described earlier without the user ever being made aware of it.

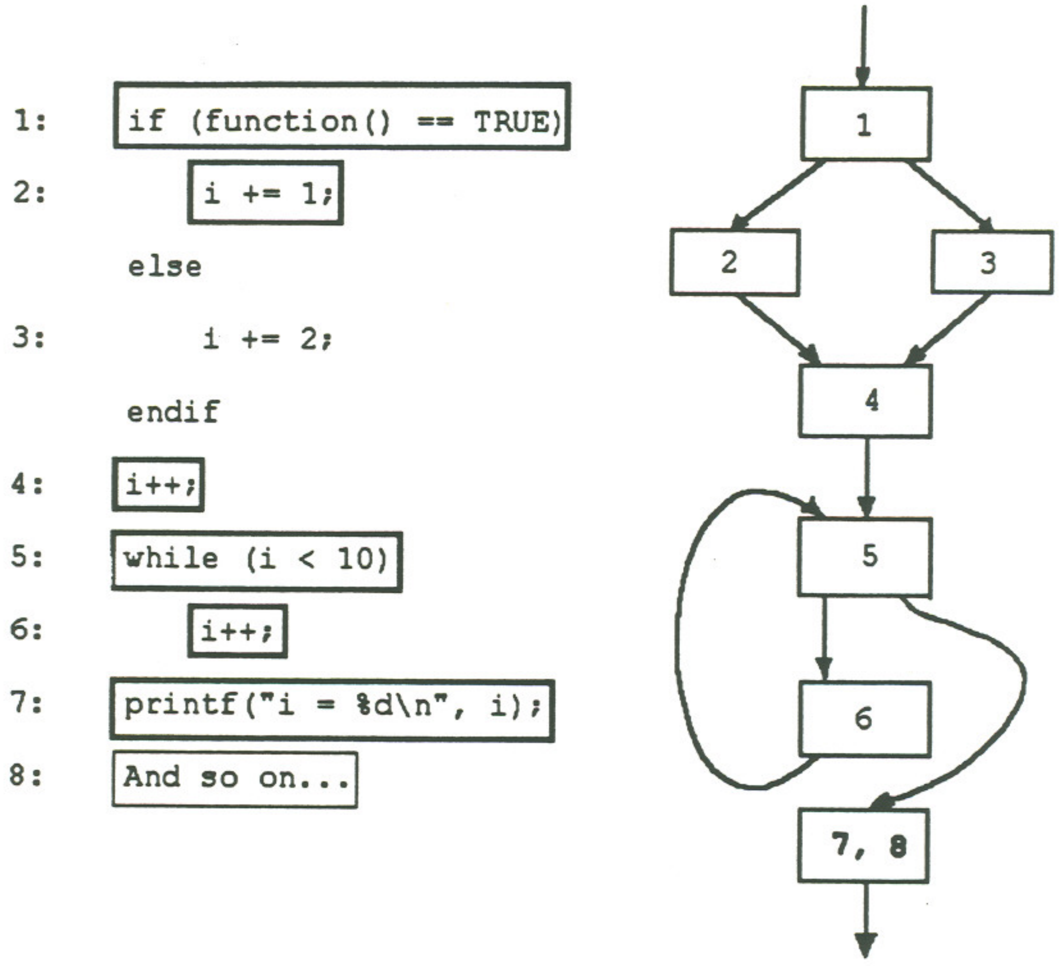
Finally, after some period of execution, the user's breakpoint will be hit leaving the time stamp entries as shown below:

Source Line	Time Stamp
1	none
2	1
3	0
4	2
5	none
6	5
7	none
8	none

Here is the sequence of actions on the part of the debugger to highlight the source (refer again to Figure 4.4):

- The expressions corresponding to line 8 will be highlighted to indicate the "current" location.
- Working backwards, line 7 will be highlighted. Continue backwards to block 5.
- Line 5 will be highlighted.
- Block 5 has 2 predecessors. The one with the highest time stamp is a backwards branch at the end of the loop. Since we haven't been this way before, follow it. (In the event that the loop contains a conditional branch, this will cause the branch taken during the most recent iteration to be highlighted.)
- Line 6 is highlighted. Continue backwards to block 5.
- Line 5 has 2 predecessors. We already followed the backwards branch, so this time, we will follow the one with the next highest time stamp to block 4.
- Line 4 is highlighted
- Block 4 has 2 predecessors. The one with the highest time stamp is block 2. Continue backwards to block 2.
- Line 2 is highlighted. Continue backwards to block 1.
- Line 1 is highlighted.

Figure 4.5 shows the result of this activity (along with the control-flow graph for clarity). This highlighted source fragment is what would be displayed in the source window of the debugger.



```
1:    if (function() == TRUE)
2:        i += 1;
      else
3:        i += 2;
      endif
4:    i++;
5:    while (i < 10)
6:        i++;
7:    printf("i = %d\n", i);
8:    And so on...
```

Figure 4.5. Source Fragment (with highlighting) and Control-Flow Graph

4.5. Implementation Of Typical Debugger Commands

4.5.1. "Go til"

Ordinary commands like "go til line 5" need to be reconsidered. Frequently, what the user actually wants is to "stop before computation x" or "stop after computation y" which may very well be a single expression in the source. Unfortunately, instruction scheduling can cause the code for one source line to be intermixed with the code for its neighbors.

The best solution to this, I believe, is to have expression (rather than line) oriented commands. In other words, "Go til" kinds of commands would be implemented as "Go til expression z" where the expression is pointed at by some means. The debugger would consult the code map and find the object code corresponding to the desired expression and plant a breakpoint. When the breakpoint is hit, the screen is painted as indicated previously to show the state of the program.

4.5.2. "Go from"

Even in non-optimized code, this is a dangerous kind of command because it is implemented essentially by a simple change of the instruction pointer. Clearly, this may violate assumptions made when code was generated, like the contents of some critical registers. If the user is careful, however, the command can yield useful results. It can be used, for example, to skip a particular source line, or to re-execute a function with different data, or to change the number of iterations of a loop.

After instruction scheduling, all of the above is still true, except that the user must exercise even greater care. Since the smallest practical item which can be skipped or repeated is an individual machine instruction, and since individual machine instructions may contain code for several source lines and loop iterations, the additional complexity compared to non-optimized code is possibly very great. The ability to highlight all of the expressions in the source corresponding to a given instruction pointer would enable the user to exercise an appropriate amount of care. Nonetheless, this is a good example of a case in which the debugger hands the user enough rope with which to hang himself.

4.5.3. "Get or Set the value of x"

Existing and proposed debuggers for optimized code attempt to hide the fact that the code is optimized [4] [20]. They will try to give the impression that the code executes on a statement-by-statement basis. For example, the user is led to believe that the source lines preceding the current one have executed to completion and succeeding ones have not been touched. If the user queries the debugger for a variable that apparently received a value in the preceding statement but in fact did not, these debuggers tend to rely on a possibly bewildering barrage of messages to indicate that fact. The "range" scheme [4] plus the highlighting scheme proposed here achieves the same effect of telling the user which variables are available, but in a much friendlier manner.

4.6. Advantages

The main advantage is that the user gets an accurate indication of exactly how the program executes, but without getting buried in an avalanche of messages.

4.7. Disadvantages

If the debugging session happens in terms of a modified source, it may not be apparent how to translate that knowledge back into original source level terms to fix the bug. In other words, suppose that the user spends a considerable amount of time looking at the modified version of the source as presented by the debugger. When it comes time to go back to the original source code to make whatever changes are necessary to fix the bug, the original source may look unfamiliar.

CHAPTER 5.
Examples

5.1. Extended example of code map usage and interpretation

Here is an extended example that is meant to illustrate how the ideas in the previous chapter work if applied to real architecture and scheduled code.

The target architecture is the Intel i860. This may seem to be a strange choice since the instruction word is not very large. However, the architecture does embody the idea of having multiple execution units controlled be single instructions which is an essential element in VLIW architectures. It is also in fairly wide use. Perhaps most importantly, though, is that we were able to use the Portland Group C compiler which is able to achieve software pipelining of many loops.

5.1.1. Architectural Overview

According to the i860 programmer's manual [13], instructions can be grouped into two "classes". The first class consists of what are called "core instructions" which are executed by the integer unit. Core instructions include integer arithmetic, loads and stores between memory and registers (including floating point registers), control transfer instructions, and others. The second class of instructions are floating-point and graphics instructions which are executed by the corresponding functional units. Many floating-point instructions come in both scalar and pipelined versions. In addition, there are instructions which manipulate both the floating-point adder and multiplier pipelines simultaneously yielding two results per clock cycle. Finally, these can be executed in parallel with instructions in the core unit.

A few notes on reading the code in the following examples:

- a "d." prefix indicates dual-instruction mode in which a floating point instruction is issued in parallel with a core instruction. The second half of the dual instruction is on the indented second line.
- instructions that start with "p" (like pfadd) are pipelined floating point instructions. An actual instruction like "pfadd r1, r2, r3" would start the calculation of r1 and r2, and would put the result of the third previous pipelined addition into r3.
- instructions following the "bla" instruction will execute in the branch delay slot. If the processor is in dual-instruction mode, the entire next dual-instruction-mode instruction will execute in the delay slot.
- the code includes instructions like "m12tpa" which is an example of a dual-operation instruction. That is, this instruction manipulates both the adder and multiplier pipelines simultaneously.
- it is possible to use the "d." prefix on dual-operation instructions resulting in dual operation instructions in dual-instruction mode (i.e., dual operation instructions in parallel with core instructions.)

5.1.2. Code Listings

Here is the source code for this extended example. Notice that each line in the body of the loop is dependent in some way upon the previous line, and each iteration of the loop is dependent upon the previous iteration. I added the line numbers in the source and the instruction offsets in the code listings so that I could construct the debug information maps.

```
1:   #define N 50
2:
3:   double a[N], b[N];
4:
5:   main()
6:   {
7:           int i;
8:
9:           a[0] = b[0] = 0;
10:          i=1;
11:         while (i < N)
12:         {
13:                 a[i] = a[i-1] + 3;
14:                 b[i] = a[i] * 5;
15:                 i++;
16:         }
17: }
```

This abbreviated code listing was produced by the compiler when no optimization was specified. The full listing is in appendix A.

```
    // lineno: 9
        orh ha%_b, r0, r31
        fst.d f0, l%_b(r31)
        orh ha%_a, r0, r31
        fst.d f0, l%_a(r31)
    // lineno: 10
        adds 1, r0, r29
        st.l r29, -4(r28)
    // lineno: 11
    .B54:   //.B0000
        ld.l -4(r28), r29
    .DB.B5454:
        adds -50, r29, r0
        bnc .B55
    // lineno: 13
        ld.l -4(r28), r29
        shl 3, r29, r30
        orh h%_a+-8, r0, r31
        or l%_a+-8, r31, r16
        fld.d r16(r30), f16
        orh ha%.C00058, r0, r31
        fld.d l%.C00058(r31), f18
        orh h%_a, r0, r31
        or l%_a, r31, r17
        fadd.dd f16, f18, f20
```

```
21:       fst.d f20, r30(r17)
    // lineno: 14
22:       ld.l -4(r28), r29
23:       shl 3, r29, r30
24:       orh h%_a, r0, r31
25:       or l%_a, r31, r16
26:       fld.d r16(r30), f16
27:       orh ha%.C00059, r0, r31
28:       fld.d l%.C00059(r31), f18
29:       orh h%_b, r0, r31
30:       or l%_b, r31, r17
31:       fmul.dd f16, f18, f20
32:       fst.d f20, r30(r17)
    // lineno: 15
33:       ld.l -4(r28), r29
34:       adds 1, r29, r30
35:       st.l r30, -4(r28)
    // lineno: 16
36:       br .DB.B5454
37:       ld.l -4(r28), r29
    // lineno: 0
    .B55:    //.B0001
```

Each loop iteration runs from the instruction at offset 9 to the instruction at offset 37; in other words, for 29 clock cycles. Notice, as I described in the last chapter, that there is a distinct block of object code for each (meaningful) source line (in the listing, look for "// lineno: some_number" for examples). This means that the mapping from source code to object code is very straightforward.

Here is manually constructed code map:

Line Number	Instruction Offset
9	2
10	6
11	8
13	11
14	22
15	33
16	36
17	38

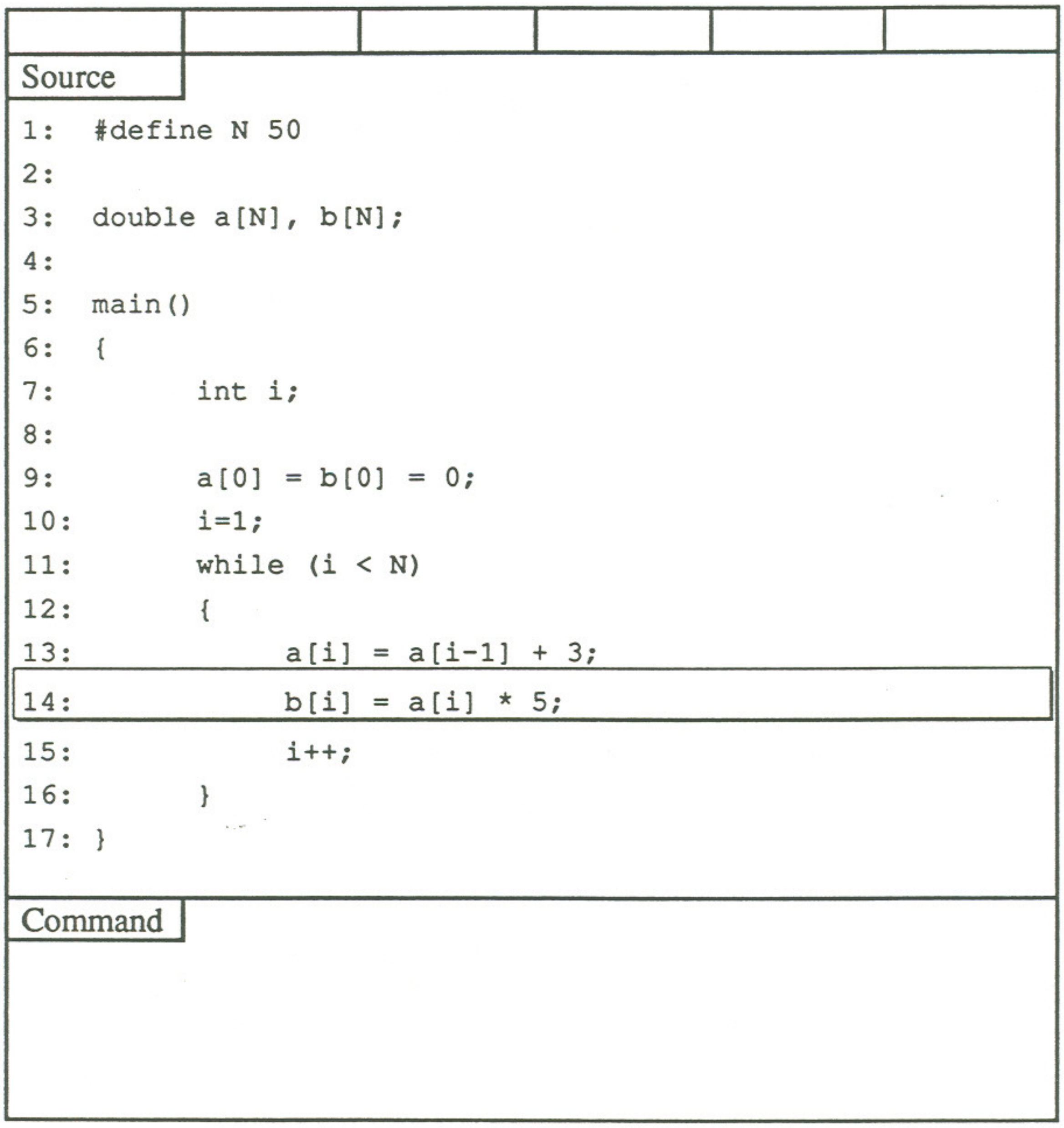
Notice also that the lines involving calculations of some kind tend to follow the same pattern. Look at the listing for line 13 as an example. It starts with some loads (and "ors" which are part of the mechanism for getting 32-bit addresses into registers), then there is an "fadd" (for floating-point add), then a "fst" (for floating-point store). The pattern is one of loading values into registers, performing the calculation, then storing back to memory. On every statement boundary, the memory location corresponding to every variable is up-to-date. This means that the mapping from variable names to their locations is very simple.

Here is a manually constructed data map:

Name	Type	Location
i	integer	stack offset*
a	array[50] of double	memory*
b	array[50] of double	memory*

*The actual value will be known at run-time

These two simple maps are all that is needed to debug the non-optimized code. Setting a breakpoint on line 14 and running until the breakpoint was hit would result in a display like Figure 5.1:



```
1:   #define N 50
2:
3:   double a[N], b[N];
4:
5:   main()
6:   {
7:           int i;
8:
9:           a[0] = b[0] = 0;
10:          i=1;
11:          while (i < N)
12:          {
13:                 a[i] = a[i-1] + 3;
14:                 b[i] = a[i] * 5;
15:                 i++;
16:          }
17:  }
```

Figure 5.1. Breakpoint Hit at Line 14: Non-optimized Code

The user would immediately know that line 14 is next; that all previous lines have executed to completion; and that line 15 has not started. With regard to querying the debugger for the value of program variables, it is immediately obvious that `i` and `a[i]` are up-to-date since they were assigned to in previous statements while `b[i]` has a spurious value since it has not been assigned to yet.

Now consider the code produced by the same compiler at the highest optimization level. This includes all optimizations performed by the compiler including software pipelining. The following is an abbreviated listing; the full listing is in appendix A. I have added comments where appropriate to clarify the code.

```
    // lineno: 0
3:  orh ha%.C00059, r0, r31
4:  fld.d l%.C00059(r31), f10   // f10 contains constant 5.0
5:  orh ha%.C00058, r0, r31
6:  fld.d l%.C00058(r31), f12   // f12 contains constant 3.0
7:  orh h%_a+-8, r0, r31
8:  or l%_a+-8, r31, r21
9:  orh h%_a, r0, r31
10: or l%_a, r31, r4        // r4 contains address of a[]
11: adds -1, r0, r16        // r16 initialized to -1
    // lineno: 9
12: orh h%_b, r0, r31
13: or l%_b, r31, r29       // r29 contains address of b[]
14: fst.d f0, 0(r29)        // initializes b[0]
15: fst.d f0, r0(r4)        // initializes a[0]
16: adds 1, r0, r30
17: adds -50, r30, r0
18: bnc .B55
19: adds 48, r0, r17        // r17 initialized to 48
20: fiadd.dd f0, f0, f8     // f8 initialized to 0
21: mov r4, r19             // r19 contains address of a[]
22: mov r29, r20            // r20 contains address of b[]
23: bla r16, r17,.B89       // loop control. manual A-3, 5-18
24: pfmul.dd f0, f0, f0
    // lineno: 13
```

```
     .B89:   //.M0000
     .align 8
25: d.pfadd.dd  f0,  f0,  f0  // starts dual instr. mode
      nop
     .PL1001:
26: d.pfadd.dd f8, f12, f0    // starts f8 (=0) plus 3.0
      nop
     .DB.B8989:
27: d.pfadd.dd f0, f0, f0     // advances apipe
      nop
28: d.pfadd.dd f0, f0, f0     // advances apipe
      nop
29: d.m12tpa.dd f10, f8, f8   // f8 gets apipe;starts f10*f8
      fst.d f8, 8(r19)++      // stores f8 in a[1]; r19 inc.
30: d.fnop
      bla r16, r17, .PL1002   // loop control
31: d.pfmul.dd f0, f0, f0     // advances mpipe
      nop
     .PL1002:                 // Now entering body of loop
32: d.pfadd.dd f8, f12, f0    // starts f8 plus 3.0
      nop
33: d.mm12msm.dd f0, f0, f30  // f30 gets mpipe from m12tpa
      fst.d f30, 8(r20)++     // stores f30 in b[]
34: d.pfadd.dd f0, f0, f0     // advances apipe
      nop
35: d.m12tpa.dd f10, f8, f8   // f8 gets apipe;starts f10*f8
      fst.d f8, 8(r19)++      // stores f8 in a[]; r19 inc.
36: d.fnop
      bla r16, r17, .PL1002   // loop control
37: d.pfmul.dd f0, f0, f0     // advances mpipe; delay slot
      nop
     .PL1003:
38: d.pfmul.dd f0, f0, f30    // f30 gets mpipe from m12tpa
      fst.d f30, 8(r20)++     // stores f30 in b[]
39: d.fnop
      bla r16, r17, .PL1003   // loop control. not needed?
```

```
40: d.fnop
      nop
41: fnop
      nop
42: fnop
      nop
    // lineno: 0
    .B55:   //.B0001
```

In this case, each loop iteration runs from the instruction at offset 32 through the instruction at offset 37, or 6 clock cycles. The fully optimized version will therefore run close to 5 times faster than the non-optimized one.

Here is a manually constructed code map:

Offset	Corresponding source	Iteration	Comments
14	line 9 assignment to b[0]		
15	line 9 assignment to a[0]		
26	start line 13 add	1	loop prolog
29	end line 13; start line 14 mult.	1	
32	start line 13 add	2	loop start
33	end line 14	1	
35	end line 13; start line 14 mult.	2	
36	conditional branch to loop start		
37	delay slot		loop end
38	end line 14	2	loop epilog
44	end of main		will return to caller

The following is a manually constructed data map. Notice that the iteration variable "i" is expressed in terms of an equation:

Name	Type	Value	IP range	Iteration
5	constant	register f10	5-end	
3	constant	register f12	7-end	
i	integer	expr (N-(r17+2))	25-end	
a	array of double	register r4	11-43	
a[0]	array element	register f8	21-29	
a[1]	array element	register f8	30-31	1
a[i-1]	array element	register f8	32-35	2
a[i]	array element	register f8	36-37	2
b	array of double	register r29	14-end	

The preceding two maps are all that the debugger gets from the compiler. Now it is time for the debugger to go to work. The first thing to do is to build the control-flow graph representation of the program. The source code and the resulting control-flow graph are shown in figure 5.2.

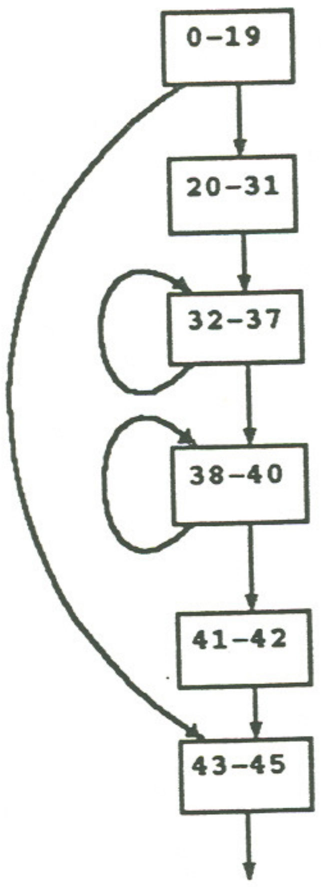


Figure 5.2. Control-Flow Graph

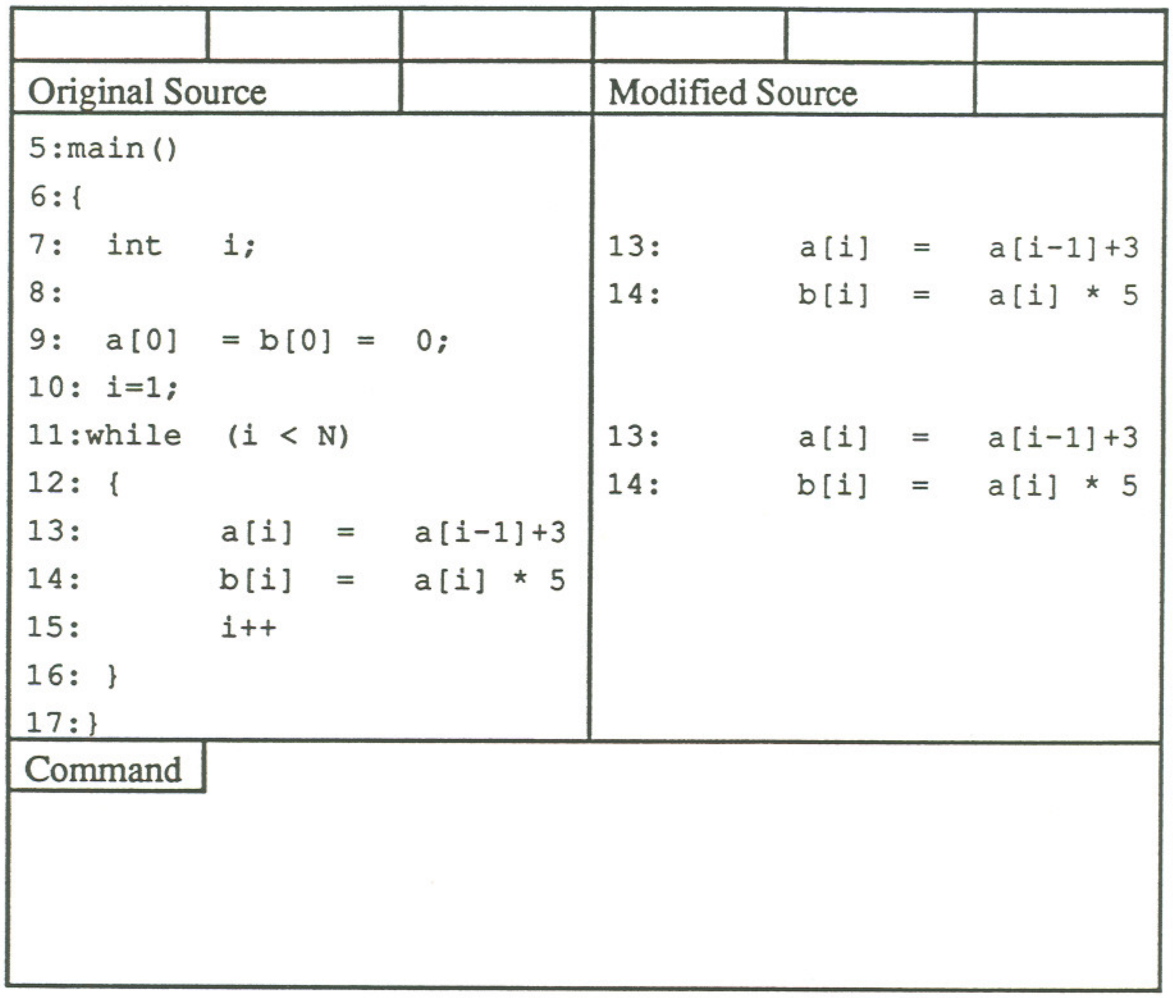
The numbers in the above boxes refer to the instruction offsets in the object code listing. The apparent gaps in the sequence are due to instructions which execute in the branch-delay slot.

Now the debugger finds all path determining instructions. In this example, the instruction at offset 32 has predecessors at offsets 30 and 36; the instruction at offset 38 has predecessors at offsets 36 and 39; the instruction at offset 43 has predecessors at offsets 39 and 18. The debugger takes this information and the code map as received from the compiler to create its own means of tracking the execution path. This involves in essence adding two more columns to the code map for the purpose of handling breakpoints. It also involves adding rows where necessary to handle path-determining instructions which do not already have entries.

This is what the debugger would work with at run-time:

Offset	Corresponding source	Iteration	T.S.	Opcode	Comments
14	line 9 assignment to b[0]		none		
15	line 9 assignment to a[0]		none		
18	conditional branch		0	bnc	
26	start line 13 add	1	none		prolog
29	end line 13; start line 14 mult.	1	none		
30	conditional branch to loop start		0	d.fnop; bla	
32	start line 13 add	2	none		loop start
33	end line 14	1	none		
35	end line 13; start line 14 mult.	2	none		
36	conditional branch to loop start		0	d.fnop; bla	
37	delay slot		none		loop end
38	end line 14	2	none		epilog
39	conditional branch		0	d.fnop; bla	
44	end of main				will return to caller

Figure 5.3 shows what might be presented to the user:



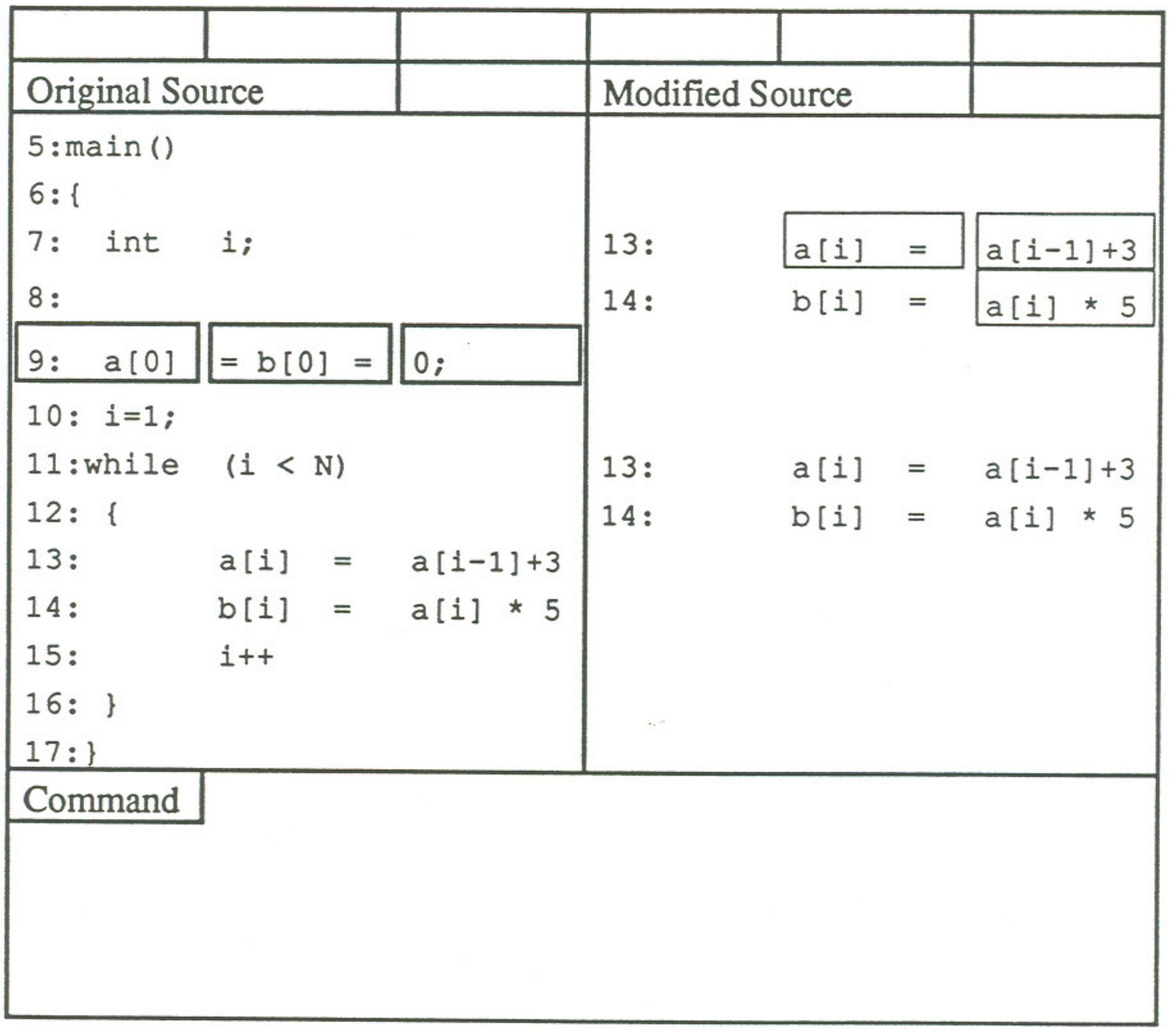
```
5:main()
6:{
7:  int   i;
8:
9:  a[0]  = b[0] =  0;
10: i=1;
11:while  (i < N)
12: {
13:         a[i]  =   a[i-1]+3
14:         b[i]  =   a[i] * 5
15:         i++
16: }
17:}
```

```
13:         a[i]   =   a[i-1]+3
14:         b[i]   =   a[i] * 5

13:         a[i]   =   a[i-1]+3
14:         b[i]   =   a[i] * 5
```

Figure 5.3. Source View Before Execution

Suppose that the user indicates the multiplication in line 14 in the original source for a breakpoint. The code map indicates that this corresponds to the instructions at offset 29 and 35. That is, the debugger interprets this as a request to break at the instructions which start the multiplication. A user breakpoint is installed at offset 29 and 35 and execution begins. The path determining instruction at offset 18 is hit. Transparently to the user, the debugger goes through all the steps outlined in the previous chapter. This involves incrementing the time stamp value, copying it to the entry for the instruction at offset 18, copying the original instruction to its location, single-stepping it, putting the breakpoint opcode back in, and continuing execution. Some time later, the breakpoint at offset 29 is hit. Now it is time for the debugger to process all of its information in order to update the screen and present it to the user. The debugger starts at the instruction at offset 29 and uses the control-flow graph to work backwards as described in the previous chapter. It highlights each expression and variable in the source according its state, i.e., not started, started, finished, or next. Figure 5.4 shows what would be presented to the user.



```
5:main()
6:{
7:  int   i;
8:
9:  a[0] = b[0] = 0;
10: i=1;
11:while  (i < N)
12: {
13:        a[i]  =  a[i-1]+3
14:        b[i]  =  a[i] * 5
15:        i++
16: }
17:}
```

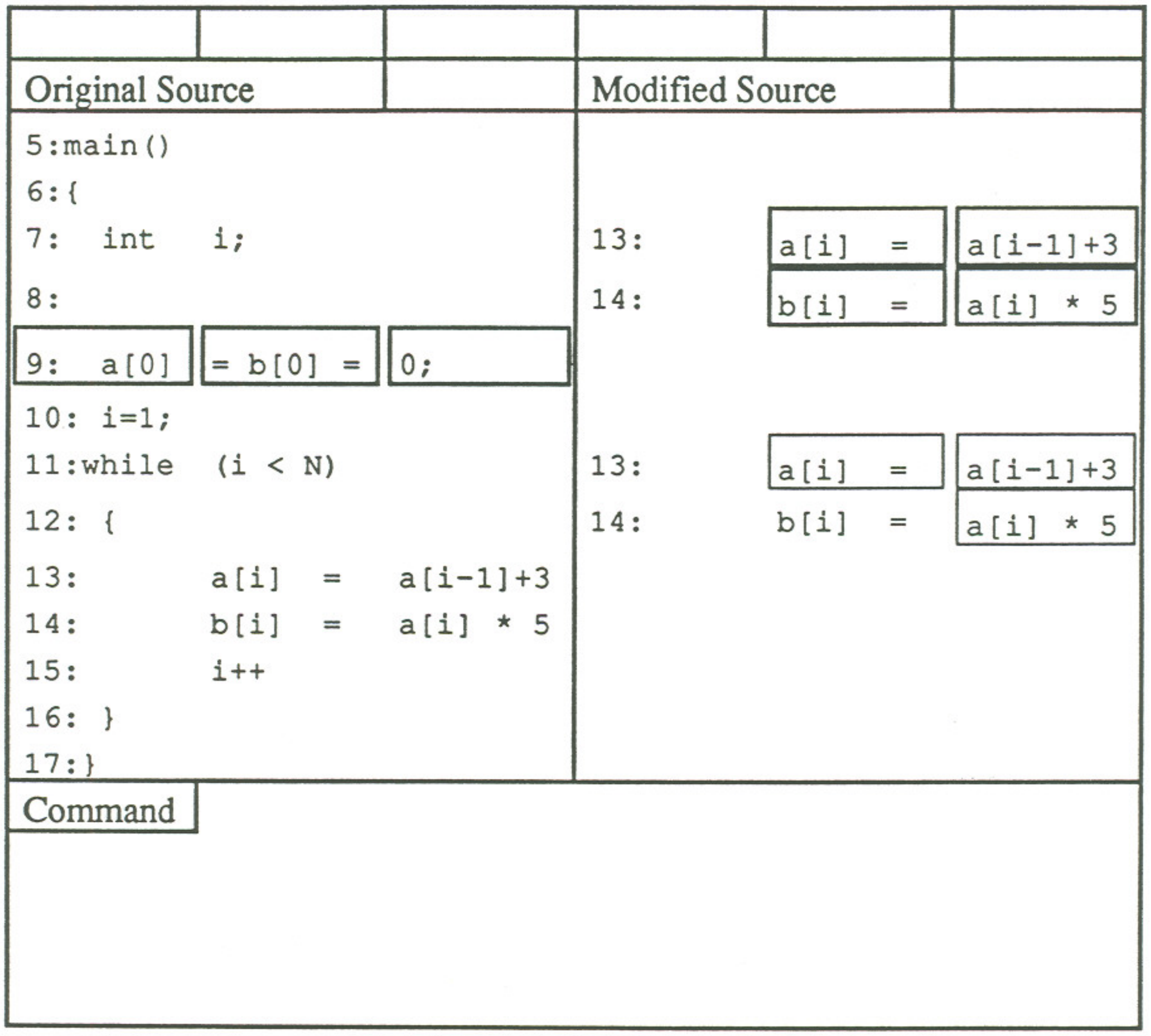
```
13:        a[i]  =  a[i-1]+3
14:        b[i]  =  a[i] * 5


13:        a[i]  =  a[i-1]+3
14:        b[i]  =  a[i] * 5
```

Figure 5.4. Source View: Breakpoint Hit At Line 14 Multiplication

Now suppose that the user continues execution. In this case, the path determining instruction at offset 30 will be hit. Again, the debugger goes through all the steps outlined in the previous chapter "behind the scenes". This involves incrementing the time stamp value, copying it to the entry for the instruction at offset 30, copying the original instruction to its location, single-stepping it, putting the breakpoint opcode back in, and continuing execution.

Finally, the user's breakpoint is hit at the instruction at offset 35. Now, updating the source is a little more interesting because we will need to go backwards through the code for two iterations instead of one. The debugger starts at the instruction at offset 35 and uses the control-flow graph to work backwards until the instruction at offset 32 which (thanks to the control-flow graph) is known to have multiple predecessors. The debugger examines the time stamps for the two path determining instructions and discovers that execution passed through instruction 30 most recently. Therefore, the debugger continues along this path, and highlights each expression and variable in the source according its state, i.e., not started, started, finished, or next. Figure 5.5 shows what the user would see at this point.



```
5:main()
6:{
7:  int   i;
8:
9:  a[0]  = b[0] =  0;
10: i=1;
11:while  (i < N)
12: {
13:        a[i]  =  a[i-1]+3
14:        b[i]  =  a[i] * 5
15:        i++
16: }
17:}
```

```
13:         a[i]  =  a[i-1]+3
14:         b[i]  =  a[i] * 5

13:         a[i]  =  a[i-1]+3
14:         b[i]  =  a[i] * 5
```

Figure 5.5. Source View: Breakpoint At Line 14 Multiplication Again

If the user continues execution until the breakpoint is hit again, the situation is only slightly more complicated. This time, the instruction at offset 36 would have a higher timestamp value than the one at offset 30. However, since the body of the loop has been handled already, the path through the instruction at offset 36 would not be followed. Instead, the debugger would proceed as the previous example explained.

5.1.3. Lessons Learned

In my hypothetical examples of the previous chapters, I didn't take into account that there might be a distinct instruction for starting a computation followed some cycles later by an instruction which completed it. I was aware that this was an issue when trying to maintain the ability to debug at the statement-level. I was surprised to see this same issue arise at the expression-level with the 860 code. One solution would be to allow the user to specify a breakpoint "before" or "after" an expression is evaluated.

This example also showed me that not every machine instruction can be "charged" to the calculation of something in the original source. I had originally envisioned that "stepping" would occur at the instruction level rather than at the statement level. However, in the presence of many instructions which do not have an obvious relation to the source, this would not be helpful. Instead, "stepping" could be implemented as executing until the next expression boundary.

5.2. A Loop with a Conditional Branch

Ordinarily, software pipelining is limited to loops containing only straight-line blocks of code. In other words, no function calls or conditional statements or other loops. However, Lam devoted quite a bit of discussion to *hierarchical reduction* [16]. With this technique, scheduling proceeded from the innermost constructs out. When a construct was scheduled, it could then be treated as if it were just a straight-line block of code with respect to scheduling the surrounding construct. This enabled her to achieve software pipelining of loops containing conditional branches, for example. This raises the question of how my proposal in the previous chapter might deal with such a situation.

For sake of simplicity, the following code is expressed in high-level language terms. The steady-state of the loop runs from offset 15 through 18.

```
0:  a[2]=a[1]
1:
2:
3:
4:
5:  b[1]=a[2]             a[3]=a[2]
6:
7:
8:
9:
10: if(i is even)         b[2]=a[3]                a[4]=a[3]
11:   c[1]=b[1]+3
12: else
13:   c[1]=b[1]-3
14:
15: d[1..n-3]=c[1..n-3]   if(i is even)            b[3..n-1]=a[4..n]   a[5..n+1]=a[4..n]
16:                         c[2..n-2]=b[2..n-2]+3
17:                       else
18:                         c[2..n-2]=b[2..n-2]-3
19:
20:                       d[n-2]=c[n-2]            if(i is even)       b[n]=a[n+1]
21:                                                  c[n-1]=b[n-1]+3
22:                                                else
23:                                                  c[n-1]=b[n-1]-3
24:
25:                                                d[n-1]=c[n-1]       if(i is even)
26:                                                                      c[n]=b[n]+3
27:                                                                    else
28:                                                                      c[n]=b[n]-3
29:
30:                                                                    d[n]=c[n]
```

Recall that for the 860 example, the instructions for each iteration of the loop would be identical. In this case, however, it is likely that during some iterations execution will go down the *if* branch of the if-else statement while during other iterations it will go down the *else* branch. This complicates the processing of path determination, but doesn't make it impossible. One solution would be to maintain a queue of the last "n" time stamps for the code in the body of the loop (where "n" is the largest degree of overlap of iterations). When it came time to update the screen, the debugger could work backwards through the code map and control-flow graph as before, but this time would go through the body of the loop once for each layer of execution as represented in the time stamps. In this way, the multiple copies of the loop body on the screen would accurately reflect which expressions had executed during which previous iteration. (The notion of keeping track of the last layer(s) of execution was mentioned briefly by Zellweger but she seemed to believe that it would require an immense amount of information.)

CHAPTER 6.
Concluding Remarks

6.1. Summary

In this paper, I have proposed that effective debugging of VLIW code after instruction scheduling can be achieved by using a combination of expression-level debug information, path determination, and highlighting in a source-level debugger to indicate the state of the program.

6.2. Strengths

Based on my research, I believe this approach has the following strengths:

- Highlighting can be used to convey complex information regarding the state of the program to the user. Other solutions for debugging optimized code rely on a barrage of messages to the user to indicate the state of the program.

- The mechanics underlying the path determination and screen highlighting should be straightforward (though certainly not trivial) to implement. Hidden breakpoints are already used in debuggers for such purposes as conditional breakpoints. Screen highlighting is already used for source-level debuggers, though only at a statement level.

- Though I focused on instruction scheduling, I believe that the approach has broad applicability to other more conventional optimizations. For example, code motion optimizations, common subexpression elimination, global register allocation, and others would seem to fall right out.

6.3. Weaknesses

It also has the following weaknesses:

- A solution along the lines of my proposal would require much more debug information than for non-optimized code. As Chapter 4 showed, going from simple non-optimized code debug information to what was required for VLIW code after software pipelining required perhaps an order of magnitude more

- There would be much more overhead in processing the debug information. Even on top of the greater amount, there is the matter of finding the right entry in the code map every time a hidden breakpoint is hit. I didn't consider what means would be used for this.

- There would be much greater difficulty for the compiler to generate this debug information. Ordinarily, compiler writers only have to worry about producing and maintaining debug information when the code is in very simple terms.

- It might take some getting used to. In comparison with an optimized code debugger that relied on informative messages to indicate the state of the program, I think the highlighting scheme would fare well. On the other hand, in comparison to a non-optimized code debugger, the user might be overwhelmed at first when confronted with a sort of "advancing checkerboard" pattern that would appear on the screen as the program advanced.

- Variable modification might be tricky. The i860 example in Chapter 5 showed that there can be several machine cycles between the start of a computation and the end. This makes the ability to modify program variables (that users of non-optimized code debugger take for granted) troublesome

- Instruction pointer modification is also a problem for the same reasons as in the previous point.

6.4. Suggestions for future work

There are many unanswered questions that arose as part of this work:

- How much debug information would be required in comparison to other "debugging optimized code" solutions as well as to non-optimized code? Also, how much would be too much?
- I mentioned that there are many possible ways to achieve path determination. An investigation into the relative efficiencies of the various techniques would be useful.
- How difficult would it be for the compiler to produce the information that would be necessary?
- How might different optimizations and combinations of optimizations be handled?

Bibliography

[1] Aho, A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] Cohn, R., Gross, T., Lam, M., Tseng, P., "Architecture and Compiler Tradeoffs for a Long Instruction Word Microprocessor", *Proceedings of ASPLOS III*, pp. 2-14, 1989.

[3] Colwell, R., Nix, R., O'Donnell, J., Papworth, D., Rodman, P., "A VLIW Architecture for a Trace Scheduling Compiler", SIGPLAN Notices 22, 10 (*Proceedings of ASPLOS II*), pp. 180-192, October 1987.

[4] Coutant, D., Meloy, S., Ruscetta, M., "DOC: A Practical Approach to Source-Level Debugging of Globally Optimized Code", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 125-134, 1988.

[5] Dehnert, J., Hsu, P., Bratt, J., "Overlapped Loop Support in the Cydra 5", *Proceedings of ASPLOS III*, pp. 26-38, 1989.

[6] Ebcioglu, K., "Some Design Ideas for a VLIW Architecture For Sequential-Natured Software", *Parallel Processing: Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing*, pp. 3-21, April, 1988.

[7] Ellis, J., *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Massachusetts, 1986.

[8] Fischer, J., "Trace Scheduling: A technique for global microcode compaction," *Transactions on Computers*, Vol. C-30, pp. 478-490, IEEE, July, 1981.

[9] Fischer, J., "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10'th Symposium on Computer Architectures*, pp. 140-150, IEEE, June, 1983.

[10] Foye, S., "Symbolic Debug of Optimized Code (SDOC)", Revision 0.0, internal Intel document, February 1991.

[11] Gibbons, P., Muchnick, S., "Efficient Instruction Scheduling for a Pipelined Architecture", *SIGPLAN Notices*, Vol. 21, No. 7, pp. 11-16. 1986.

[12] Hennessy, J., "Symbolic Debugging of Optimized Code", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 323-344, July 1982.

[13] *i860 64-Bit Microprocessor Programmer's Reference Manual*, Intel Corporation, 1990.

[14] Iyengar, S., "Symbolic Debugging of Optimized Code: A 'White' paper", Revision 1.0, internal Intel document, October 1991.

[15] Jones, R., Allan, V., "Software Pipelining: An Introduction", *Microarch*, Vol. 5, No. 2, pp. 1-4, November 1990.

[16] Lam, M., *A Systolic Array Optimizing Compiler*, PhD thesis, Carnegie Mellon University, 1987.

[17] Lam, M., "Instruction Scheduling for Superscalar Architectures", *Annu. Rev. Comput. Sci.* 4:173-201, 1990.

[18] Maier, D., Oregon Graduate Institute, Private communication, 1991.

[19] Pollock, L., Soffa, M., "High-Level Debugging With The Aid Of An Incremental Optimizer", *Proceedings of the Twenty-First Annual Hawaii International Conferences on Systems Sciences*, Vol II, January, 1988.

[20] Schlaeppi, H., Warren, H., "Debugging System Compatible With Optimizing Compiler", *IBM Technical Disclosure Bulletin*, Vol. 22, No. 6, pp. 2578-2583, November 1979.

[21] Scollard, B., "Symbolic Debugging of Optimized Programs", internal Intel document, April, 1989.

[22] Srivastava, A., "Recovery of Noncurrent Variables in Source-level Debugging of Optimized Code", *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference Proceedings*, pp. 36-56, 1986.

[23] Touzeau, R., "A Fortran Compiler for the FPS-164 Scientific Computer", *SIGPLAN Notices*, Vol. 19, No. 6, pp. 48-57, June, 1984.

[24] Wall, D., Srivastava, A., Templin, R., "A Note on Hennessy's 'Symbolic Debugging of Optimized Code'", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 176-181, January 1985.

[25] Warren, H., "No Source Change Mode For Optimizing Compilers", *IBM Technical Disclosure Bulletin*, Vol. 22, No. 6, pp. 2576-2577, November 1979.

[26] Warren, H., Schlaeppi, H., "Design of the FDS interactive debugging system", IBM Research Report RC7214, July 12, 1978.

[27] Weiss, S., Smith, J., "A Study of Scalar Compilation Techniques for Pipelined Supercomputers", *ACM Transactions on Mathematical Software*, Vol. 16, No. 3, pp. 223-245, September 1990.

[28] Weiss, S., "Scalar Supercomputer Architecture", *Proceedings of the IEEE*, Vol. 77, No. 12, pp. 1970-1982, December 1989.

[29] Zellweger, P., "An Interactive High-Level Debugger for Control-Flow Optimized Programs", *ACM SIGPLAN Notices*, Vol. 18, No. 8, pp. 159-171, August 1983

[30] Zellweger, P., *Interactive Source-Level Debugging of Optimized Programs*, Ph.D. Dissertation, Computer Science Division -- EECS, University of California, Berkeley, CA, 1984

[31] Zurawski, L., Johnson, R., "Debugging Optimized Code With Expected Behavior", pre-publication draft, University of Illinois at Urbana-Champaign, April 1991.

Appendix A

This is the full code listing for the extended example in chapter 5 which was produced by the compiler at the lowest optimization level.

```
             .file    "g.c"
         // PGC Rel 2.0 -opt 0
             .text
             .globl   _main
             .align   8
         _main:
         .a1 = 0
         .f1 = 32
0:           rh h%.STACK+.f1-16, r0, r28
1:           or l%.STACK+.f1-16, r28, r28
         // lineno: 9
2:           orh ha%_b, r0, r31
3:           fst.d f0, l%_b(r31)
4:           orh ha%_a, r0, r31
5:           fst.d f0, l%_a(r31)
         // lineno: 10
6:           adds 1, r0, r29
7:           st.l r29, -4(r28)
         // lineno: 11
         .B54:    //.B0000
8:           ld.l -4(r28), r29
         .DB.B5454:
9:           adds -50, r29, r0
10:          bnc .B55
         // lineno: 13
11:          ld.l -4(r28), r29
```

```
12:     shl 3, r29, r30
13:     orh h%_a+-8, r0, r31
14:     or l%_a+-8, r31, r16
15:     fld.d r16(r30), f16
16:     orh ha%.C00058, r0, r31
17:     fld.d l%.C00058(r31), f18
18:     orh h%_a, r0, r31
19:     or l%_a, r31, r17
20:     fadd.dd f16, f18, f20
21:     fst.d f20, r30(r17)
    // lineno: 14
22:     ld.l -4(r28), r29
23:     shl 3, r29, r30
24:     orh h%_a, r0, r31
25:     or l%_a, r31, r16
26:     fld.d r16(r30), f16
27:     orh ha%.C00059, r0, r31
28:     fld.d l%.C00059(r31), f18
29:     orh h%_b, r0, r31
30:     or l%_b, r31, r17
31:     fmul.dd f16, f18, f20
32:     fst.d f20, r30(r17)
    // lineno: 15
33:     ld.l -4(r28), r29
34:     adds 1, r29, r30
35:     st.l r30, -4(r28)
    // lineno: 16
36:     br .DB.B5454
37:     ld.l -4(r28), r29
    // lineno: 0
    .B55:   //.B0001
    // lineno: 17
38:     bri r1
39:     nop
        .data
        .align  8
    .C00058:     // (0)
        .long    0x0, 0x40080000 // 3.00000000000000000E+00
    .C00059:     // (8)
        .long    0x0, 0x40140000 // 5.00000000000000000E+00
        .lcomm   .STACK,32
        .data
        .comm    _a,400
        .comm    _b,400
```

This is the full code listing for the extended example in chapter 5 which was produced by the compiler at the highest optimization level.

```
        .file    "g.c"
    // PGC Rel 2.0 -opt 4
        .text
        .globl   _main
        .align   8
    _main:
    .a1 = 0
    .f1 = 32
0:  orh h%.STACK+.f1-16, r0, r28
1:  or l%.STACK+.f1-16, r28, r28
2:  st.l r4, -8(r28)
    // lineno: 0
3:  orh ha%.C00059, r0, r31
4:  fld.d l%.C00059(r31), f10    // f10 contains constant 5.0
5:  orh ha%.C00058, r0, r31
6:  fld.d l%.C00058(r31), f12    // f12 contains constant 3.0
7:  orh h%_a+-8, r0, r31
8:  or l%_a+-8, r31, r21
9:  orh h%_a, r0, r31
10: or l%_a, r31, r4        // r4 contains address of a[]
11: adds -1, r0, r16        // r16 initialized to -1
    // lineno: 9
12: orh h%_b, r0, r31
13: or l%_b, r31, r29       // r29 contains address of b[]
```

```
14: fst.d f0, 0(r29)       // initializes b[0]
15: fst.d f0, r0(r4)       // initializes a[0]
16: adds 1, r0, r30
17: adds -50, r30, r0
18: bnc .B55
19: adds 48, r0, r17       // r17 initialized to 48
20: fiadd.dd f0, f0, f8    // f8 initialized to 0
21: mov r4, r19            // r19 contains address of a[]
22: mov r29, r20           // r20 contains address of b[]
23: bla r16, r17,.B89      // loop control. manual A-3, 5-18
24: pfmul.dd f0, f0, f0
    // lineno: 13
    .B89:   //.M0000
    .align 8
25: d.pfadd.dd  f0,  f0,  f0  // starts dual instr. mode
      nop
    .PL1001:
26: d.pfadd.dd f8, f12, f0    // starts f8 (=0) plus 3.0
      nop
    .DB.B8989:
27: d.pfadd.dd f0, f0, f0     // advances apipe
      nop
28: d.pfadd.dd f0, f0, f0     // advances apipe
      nop
29: d.m12tpa.dd f10, f8, f8   // f8 gets apipe;starts f10*f8
      fst.d f8, 8(r19)++      // stores f8 in a[1]; r19 inc.
30: d.fnop
      bla r16, r17, .PL1002   // loop control
31: d.pfmul.dd f0, f0, f0     // advances mpipe
      nop
    .PL1002:                  // Now entering body of loop
32: d.pfadd.dd f8, f12, f0    // starts f8 plus 3.0
      nop
33: d.mm12msm.dd f0, f0, f30  // f30 gets mpipe from m12tpa
      fst.d f30, 8(r20)++     // stores f30 in b[]
34: d.pfadd.dd f0, f0, f0     // advances apipe
```

```
      nop
35: d.m12tpa.dd f10, f8, f8    // f8 gets apipe;starts f10*f8
      fst.d f8, 8(r19)++       // stores f8 in a[]; r19 inc.
36: d.fnop
      bla r16, r17, .PL1002    // loop control
37: d.pfmul.dd f0, f0, f0      // advances mpipe; delay slot
      nop
    .PL1003:
38: d.pfmul.dd f0, f0, f30     // f30 gets mpipe from m12tpa
      fst.d f30, 8(r20)++      // stores f30 in b[]
39: d.fnop
      bla r16, r17, .PL1003    // loop control. not needed?
40: d.fnop
      nop
41: fnop
      nop
42: fnop
      nop
    // lineno: 0
    .B55:   //.B0001
    // lineno: 16
43: ld.l -8(r28), r4
44: bri r1
45: nop
        .data
        .align  8
    .C00059:    // (0)
        .long   0x0, 0x40140000 // 5.0000000000000000E+00
    .C00058:    // (8)
        .long   0x0, 0x40080000 // 3.0000000000000000E+00
        .lcomm  .STACK,32
        .data
        .comm   _a,400
        .comm   _b,400
```