THESIS

REGISTER ALLOCATION AND ASSIGNMENT IN A RETARGETABLE MICROCODE COMPILER USING GRAPH COLORING

Submitted by

Steven J. Beaty

Department of Computer Science

In partial fulfillment of the requirements

for the degree of Masters of Science

Colorado State University

Fort Collins, Colorado

Summer 1987

COLORADO STATE UNIVERSITY

February 7, 1994

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY STEVEN J. BEATY ENTITLED REGISTER ALLOCATION AND ASSIGNMENT IN A RETARGETABLE MICROCODE COMPILER USING GRAPH COLORING BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTERS OF SCIENCE.

Committee on Graduate Work

____

____

____

____

Adviser

____

Department Head

ABSTRACT OF THESIS

REGISTER ALLOCATION AND ASSIGNMENT IN A RETARGETABLE MICROCODE COMPILER USING GRAPH COLORING

This thesis' focus is to investigate the retargetability of a method for allocation and assignment of registers in the Horizon compiler. This process breaks into two parts. Allocation involves assigning symbolic registers to operations that require destination resources within the machine. Assignment defines the process whereby the symbolic registers are mapped onto actual machine resources. This requires ascertaining whether enough of a specific type of resource for all the symbolic references exists. If not, some code must be added to move the symbolic resources in conflict to actual machine resources that have spare room.

The work for this thesis was done during the development of the compiler and therefore should not be considered static. However, it works well in its current configuration and is not expected to change much in the future.

Steven J. Beaty
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 1987

ACKNOWLEDGEMENTS

I would like to thank Mike Duda for his work on the compiler and his help during all phases of the implementation. I would like to thank Phil Sweany for his undying patience and his great technical ability. I would like to thank Peg Sweany for reading and correcting what had to be a boring thesis for someone who is not even remotely interested in its content. Thank you also to Ron Firooz who helped on the first version of the coloring algorithm. I will always be indebted to Bob Mueller for the chance to work on such a project and his encouragement and guidance through it all. My deepest gratitude goes to my wife, Paula, who put up with me during this process and gave invaluable support and who also read this thesis more times than I would wish on anyone.

I would also like to thank the National Science Foundation for Grants MCS-8107481 and DCR-8503941, the Amoco Foundation and the Evans and Sutherland Corporation for their support.

DEDICATION

To those who never thought I would even attend college, your memory drives me further onward.

CONTENTS

1 Introduction and Motivation **1**

2 Useful Definitions **5**
2.1 Compilers 7
2.1.1 The General Compilation Process 8
2.1.2 The Horizon Compiler 8
2.1.3 Horizon Retargeting Features 11

3 Survey of Existing Methods **13**
3.1 Branch and Bound 13
3.2 Non-homogeneous Register Banks 14
3.3 Table Driven 15

4 The Graph Coloring Method **18**
4.1 Introduction 18
4.2 Building the Interference Graph 18
4.3 An Exhaustive Method of Graph Coloring 21
4.4 The Heuristic Approach 21
4.5 Spilling 23

5 Algorithms + Data Structures **25**
5.1 Introduction 25
5.2 Graph Building in the Horizon System 25
5.3 The Implementation of the Graph Coloring Algorithm 27
5.4 Spilling 34

6 Results **36**
6.1 Introduction 36
6.2 Example #1 36
6.3 Example #2 37
6.4 Example #3 37

7 Summary and Direction for Future Work **42**
7.1 Extensions to the Graph Coloring Approach 42
7.2 Ease of Retargetability 43

8 REFERENCES **45**

LIST OF FIGURES

2.1 A Graphic Example of Live Tracks 6

2.2 Generic Compiler Outline 9

4.1 A Pathological Graph 22

5.1 Interference Graph Data Structure 27

5.2 An Example Basic Block Showing Live Tracks 29

5.3 The Interference Graph for the Example 30

5.4 The Initial Interference Graph Data Structure for the Example 31

5.5 The Graph Data Structure after One Iteration 32

5.6 The Graph Data Structure after Two Iterations 33

6.1 Example Program #1 37

6.2 Output of Coloring Algorithm for Example Program #1 38

6.3 Example Program #2 38

6.4 Output of Coloring Algorithm for Example Program #2 39

6.5 Example Program #3 39

6.6 Output of Coloring Algorithm for Example Program #1 41

Chapter 1

INTRODUCTION AND MOTIVATION

During the code generation by a compiler for a given architecture, mappings must be made from the programmer-defined operations and resources to actual operations and resources that exist on the target architecture. This process includes actions such as supplying multiplexers with data from a bus, invoking the proper ALU operation, and the topic of this paper: mapping the user-defined and compiler-generated variables to machine resources. These mappings alleviate the programmer's burden of making the best use of the registers available in the architecture. Methods exist that can produce mappings at least as efficient as a human programmer produces, given an unbounded amount of time. Ideally, optimal processes would be used since they use the fewest number of machine registers possible. However, since optimal register assignment is computationally intractable in the worst-case sense (see [Set75]), this approach proves impractical. Fortunately, close approximations of these time-consuming processes can be made that produce near-optimal results in reasonable amounts of time.

In another vein, it is possible that the first time a programmer writes a piece of code, that person could manage all bindings of the variables to the machine resources as well as a good optimizing compiler. However, during modifications, it is unlikely the programmer would change the initial assignments to better fit any changes made. The compiler, on the other hand, always recomputes the best possible assignment during each recompilation so the results remain as optimal as the first time.

Other methods of mapping the various other program operations to the machine resources of the target architecture are well-documented ([AU77],[ASU86]). This thesis reviews several methods of register allocation and assignment and discusses an implementation based on the

graph coloring method proposed by Chaitan ([CAC+81],[Cha82]). Register allocation, during compilation, chooses from an unbounded pool of symbolic registers those variables that will reside in actual machine registers. The assignment process then maps the chosen variables onto specific locations within the finite number of machine registers. The register allocation and assignment process occurs in four phases:

- *code selection*: uses an unbounded number of symbolic registers during the early parts of compilation to represent the program variables and compiler-generated temporaries. A unique name usually represents each symbolic register, generally an increasing numerical sequence. This permits the most flexibility in the resulting intermediate language. It also permits any optimization procedures (if present) to produce the best code possible because the procedures are not limited by the choices of any register mappings made by an earlier routine.
- *allocation*: chooses which of the unbounded number of symbolic registers will reside in actual machine registers.
- *assignment*: maps the chosen symbolic registers to actual hardware registers.
- *spilling*: occurs when there is an insufficient number of hardware registers to cover symbolic register requirements. Code is added to place a registers' contents, which is in contention with other registers, to memory and then back into a register when needed again. This process creates a free register that can hold another variable.

The reasoning behind the efforts of finding the best register assignment possible is that off-chip memories are much slower than registers; historically 10:1, now at best 3:1–2:1. Therefore, any operations performed with the values in the machine's registers will execute at least 2–3 times faster than the same operations done with the values residing in some external memory. This implies that code that uses registers well can have speedups of 2–3 times that of code that does not use them well. Also, most modern machines have many registers. Using these resources as well as possible exploits this commitment of silicon.

This paper's context should be noted to reduce confusion. N. Tredennick [Tre82] enumerates various "cultures" of microprogramming:

- *The commercial machine culture* uses microprogramming to implement a single instruction set on a variety of differing machines. The IBM System/370 is a good example where machines with various internal hardware are microprogrammed to accept the same "high-level" assembler language.
- *The bit-slice culture* uses generic bit-slice components and are programmed to perform one specific task. These machines are generally programmed by their designers so that high performance is achieved. However, there currently is interest in having the end-user program these machines to better match the machine and its intended application.
- *The microprogrammable machine culture* builds machines that are intended to execute native code for various architectures by modifying (at run time) the microcode control store. The Nanodata QM1 is an example of this type of machine.
- *The single-chip culture* has a control store (usually a Read Only Memory) on the chip which is microprogrammed. A processor of this type is microcoded as an implementation technique and the microcode usually manages the low-level register–to–register and register–to–functional-unit transfers. The Motorola 68000 series chip are an example of this type of "microprogramming".

Register assignment (and compilation itself) makes little sense in all but the bit-slice culture. The other three cultures are based on implementing an instruction set in the target machine. This generally requires short sequences of microcode independent from one another. All operations to be performed must fit into the available registers because it is either impossible or very costly (in terms of time required) to spill variables. In the bit-slice culture however, there are no instructions to implement. The need is to implement an algorithm to carry out a specific task. This need usually results in long sequences of code that are dependent upon each other. In this case, compilers can

take a high-level description of the algorithm and produce well-compacted code to be run on the target architecture. In addition to a writable microcode store, secondary scratch memories generally exist within these micro-architectures allowing for reasonable spilling costs.

Historically, various register assignment methods produced less than optimal results. Global register assignment by compilers always uses a specified register for a specified purpose. For example, using address registers A1 and A2 for only address calculations involving two dimensional arrays limits their usefulness. This method will probably perform poorly, especially when the vast majority of code produced does not involve two dimensional arrays and these two registers are unavailable for use in other computations.

Another method that produces less optimal results than current methods is assignment during code selection. In this process, when a register is needed for a computation, a check is made for a free register. If one is not available, a heuristic spills a register, such as spilling the register that has been used least recently. Unfortunately, this method does not take into account global data flow information (such as the next time a variable will be used), and therefore it is likely to make bad decisions regarding which variables to give registers and which ones to spill. Several algorithms that produce better assignments will be discussed, followed by a more in-depth discussion on the graph coloring method. But first, some useful definitions are introduced.

Chapter 2

USEFUL DEFINITIONS

Some definitions referenced throughout this thesis which pertain to code in general and microcode specifically follow:

- A variable is *live* at some program point Q if and only if the variable is used at program points reachable from Q. We say that definition D *reaches* a program point P if a path exists from the program point immediately following D to P, such that D is not killed along that path. For example, in the following program the variable A is live at least at points 2, 3 and 4:

  [1] B := 0;

  [2] A := 1;

  [3] C := 2;

  [4] D := A + 3;

- The *live track* (or simply *track*) of a variable is the set of consecutive program points that the variable is live. In the above example, the live track for A is [2–4]; also see Figure 2.1.

- The first point in a track is the *definition point*. Any points where a variable is used, including the last point, is a *use point*. Any program point where a live variable unused is known as a *dormant point*. (Note: A live track might have dormant subtracks.)

- Assignment rule: two variables live at the same program point cannot be assigned to the same register at that point. This is the basis for the need for register assignment.

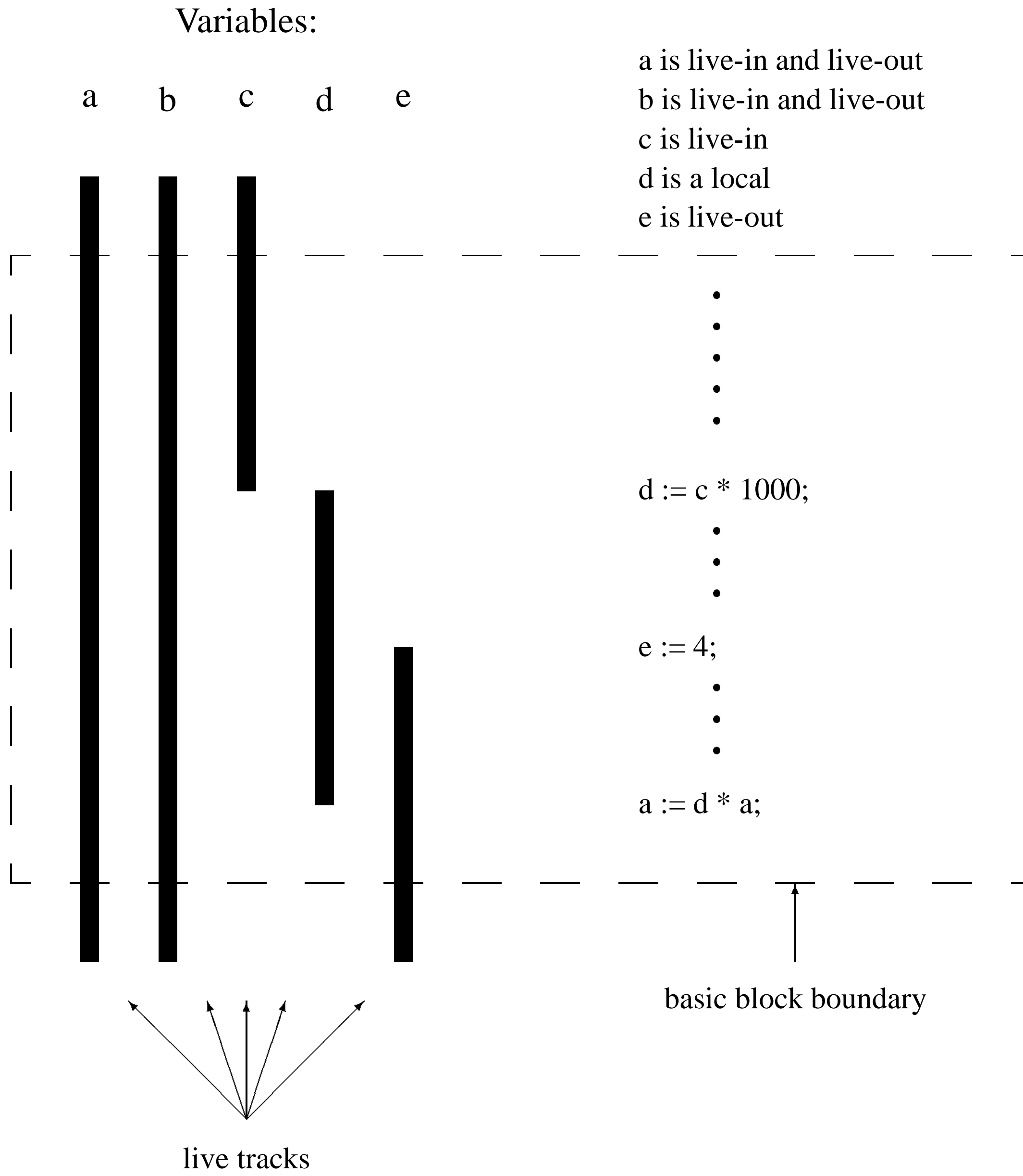

Figure 2.1: A Graphic Example of Live Tracks

- Two tracks are *compatible* if they do not break the assignment rule; i.e., they are disjoint.
- A *basic block* is a sequence of code with only one entry and one exit point.
- An operation O is *data dependent* on another operation P if O reads a value written by P. This implies that the code performing the operation for O must follow in its execution the code that performs the operation for P.
- An operation O is *data anti-dependent* on another operation P if O writes to a location that needs to have been previously read by P. This implies that the code performing the operation for O must follow in its execution the code that performs the operation for P. In this example, [2] is data dependent on [1] and [3] is data anti-dependent on [2].

  [1] A := 1;

  [2] D := A + 3;

  [3] A := 1000;

  With these code restrictions, no other ordering would preserve the meaning of this code sequence. See [Veg82].
- In microcode, primitive machine operations that are called *micro-operations* (MOs) are packed into *micro-instructions* (MIs).

For a broader overview of current fundamentals in microprogramming and firmware engineering, see [MV88].

2.1 Compilers

A compiler takes, as input, a sequence of high-level language constructs and produces, as output, source code for a given *targeted* architecture. It, thereby, provides the user a level of abstraction which, hopefully, removes the need for the user to have a deep understanding of the targeted architecture. A compiler is considered *retargetable* if it can, with a reasonable amount of effort, compile code for various machine architectures. A compiler is considered to be *optimizing*

if it performs transformations on the input code that reduces either the length of run-time execution or the size of the run-time code. Note it may, in general, be difficult to ascertain from the resulting machine-level code the high-level language source that produced it. The optimizations may move code to places distant from their original declaration in the high-level language.

2.1.1 The General Compilation Process

In general, the compilation process proceeds as follows. First, a lexical analyzer scans the input code, removes the white space from the file, and breaks the remaining input into tokens. The tokens are then passed to a parser which verifies that the program is syntactically correct. Depending on the implementation, the parser may also build various symbol tables containing all of the declared program attributes. Expression trees can also be built from the expressions in the source code. The parser then builds either an intermediate representation of the code or, in simple non-optimizing compilers, generates the target code as it parses. In optimizing compilers, additional processes then operate on the generated intermediate code. Code-improving operations (such as constant folding, loop-invariant hoisting and common subexpression elimination) can be performed when the global information found in the intermediate representation is available. Data-flow information, such as which variables are live-in and live-out of a block, can also be deduced. This allows for even more optimizations to be performed, such as dead-code removal and, of course, intelligent register assignment. At any stage in the compilation process, peephole optimizers can be introduced to do simple code transformations such as removing unneeded compiler-generated temporaries that exist between a source and a destination. See Figure 2.2 for a graphical overview of a typical compiler.

2.1.2 The Horizon Compiler

Currently the Horizon compiler parses a subset of the C programming language known as Micro-C. This is a representation unlike other attempts at microcode representation languages (which are also known as Micro-C) and should not be confused with them. See ([DM84],[DMa],[DMb]) for discussions on the actual format and implementation of the language. The lexical analyzer and parser produce an internal representation based on

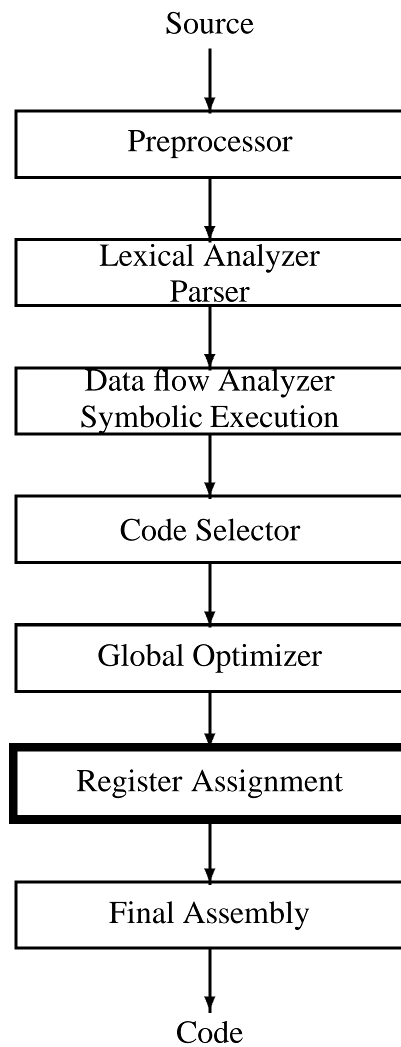

Figure 2.2: Generic Compiler Outline

“ssm’s.” (“ssm” is a holdover from earlier work and has no underlying meaning in the current implementation.) A peephole optimizer currently runs on this form of the program to remove such inefficiencies as unneeded temporaries generated by traversing the expression trees. Note that while this operation could be performed during expression-tree traversal, it is much less complex and less machine-dependent to do this using a peephole method. It is hoped that the peephole optimizer will be one of the few machine-dependent routines in the front-end. There currently appears to be no way to produce a good intermediate language representation from expressions in a completely machine-independent way because of machine-specific issues such as whether or not the ALU’s are two-address or three-address. After peephole optimization, a global control-flow graph is generated to provide useful information for the later stages.

Data-flow analysis is performed by making calls to the code selector. This is a new way of computing the live-in and live-out sets for a block. But, since the operations to compute these sets are already in the code selector, there is no need to provide separate, and redundant, data-flow routines. Cover analysis is then done to provide information similar to reaching definitions so that the extents of code motion can be found. Cover analysis provides the highest and lowest points in a program where a specific instruction may be placed. Common subexpression elimination, which requires the results from cover analysis, is then provided. Memory disambiguation determines when memory-based variable tracks may overlap, as suggested in [Ell86]. Disambiguation, especially valuable in distinguishing between references of the same array, requires the information that is provided by covers.

After the intermediate program form is produced, a code selector is called with each ssm to produce a corresponding code sequence for the target machine [BDMW86]. Then, the code sequences are repeatedly generated for each ssm and merged. This merging is constrained both by the data dependencies inherent in the code and by the resource usages between code sequences. The series-parallel coupler routine does this important and pervasive process (see [MS86] for an in-depth discussion). After all code sequences are generated and merged, a routine coalesces nodes that have no data dependencies and no resource timing conflicts. Two numbers specify resource timings between nodes: the data-dependent min and max times. The data-dependent min time

specifies how many MI cycles must elapse before a resource can be read after it is written. The data-dependent max time specifies for how many MI cycles after a resource is written it can be read. Registers generally have $(1, \infty)$ timing because they are written at the end of the current cycle and, therefore, cannot be read until the next cycle. After that, they may be read at any time because their values are not transient. Nodes may be coalesced if they are not data dependent upon one another and if they have $(0, 0)$ resource timing between them. The output of all the code sequences being merged together is a data-dependency graph (DDG) representing the program's inherent ordering. The DDG is represented as a directed acyclic graph (DAG), and the two acronyms will be used interchangeably.

After the merging phase has been completed, register assignment (the topic of this paper) is performed. Loop unrolling, specifically URCR (of which URPR [SDX86] is an extension), is then performed by copying a DAG for a loop and then serially merging the two copies of the DAG together. Trace scheduling, suggested in [Fis81] and demonstrated in ([GS82],[GS83],[SDJ84]), is then done, again using the series-parallel coupler across block boundaries. Finally, Local compaction is performed on a block-by-block basis, and the final assembler-ready code is output. See [All86] for a general overview of the Horizon compilation process.

Currently, the Horizon compiler divides into two distinct phases: a machine-independent front-end and a machine-dependent back-end. As noted before, the front-end appears to need a limited amount of machine-dependent information, but this is kept to a bare minimum. The code selector marks the beginning of the back-end. The back-end is coupled together by a phase-control process intended to be flexible by allowing a polynomial description to be made of the interactions between the various phases of the back-end [All86]. An overview of the Horizon system, with a discussion of retargeting in particular, can be found in [MDSW86]. See Figure 2.3 for an abstract overview of the current compiler.

2.1.3 Horizon Retargeting Features

The Horizon compiler can target its code-selection process to a wide variety of target architectures. The basic machine must have a single control store, a single micro-instruction word,

and it must operate in a synchronous fashion. Such machines have been referred to as "clocked micro-architectures" in [Das84]. With these restrictions in mind, the target machine may have (from [Plo87]):

- An arbitrarily wide control word,
- Polyphase or monophase execution models,
- Pipelined fetch and execute cycles,
- SIMD and/or pipelined functional units.
- Permanent or transient storage elements with arbitrary discrete setup and hold times,
- Direct, indirect, bit steering or format shifting forms of control word encodings,
- Micro-operations with side effects,
- Delayed branching schemes.

The Horizon compiler achieves the goals both of producing highly optimized code and of being retargetable by choosing machine resource usage information as the primary source of machine-dependent input. In the back-end, information is required to denote how information is passed from one functional unit to another, what the timings for the various resources are, and how the functional units transform their inputs to form their outputs. This information is described by a fairly high-level machine-description language based on a vestige from an earlier version of the Horizon compiler. Specifically, they look like Prolog structures because an early version of the compiler was written in Prolog. For descriptions of this process see ([MW86],[Plo87]).

Chapter 3

SURVEY OF EXISTING METHODS

This section will discuss several methods of register allocation and assignment: one that is computationally complex, one that produces good results for architectures that have non-homogeneous register banks, and one that uses tables to generate the needed register assignments. Much of this information is from [OHMu84] and the original sources as noted.

3.1 Branch and Bound

The branch and bound method was one of the earliest methods of register assignment that was proposed by D.J. DeWitt [DeW76]. In his system, which compiled a high-level language into microcode, he treated code generation as a scheduling problem, similar to those problems previously studied to solve scheduling difficulties found in operating systems. DeWitt correctly points out that register assignment can have a significant impact on code generation, particularly in producing the largest amount of concurrency by the allocation order. DeWitt attempted to incorporate feedback from the register assignment phase to the code generator to overcome this problem.

Each source program operation that needs to be executed is viewed as a task that needs to be scheduled. The various data dependencies and anti-dependencies impose a partial ordering of the tasks. To achieve a final ordering, the following method is used. All possible orderings of the given partial orderings are examined. Based on some defined criteria, infeasible orderings are thrown out (bound). Then the remaining possibilities are tried by branching along all the possible paths. This is repeated until all branches are followed; that is, until the entire program has been analyzed.

This process can obviously lead to vast amounts of testing and put the problem into the computationally intractable realm. Heuristics prevent this from happening. Paths may not be

followed when their costs rise above a pre-defined level or become longer than a path already traversed. This approach does allow for spilling ahead of when spilling is actually needed, which might allow more variables to be in registers at a critical point in a program. The program could also be broken up (for example, on basic block boundaries) and then run through the process, limiting the amount of branching done.

3.2 Non-homogeneous Register Banks

Kim and Tan [KT79] developed a method that has been successfully used in architectures with non-homogeneous registers. These include architectures that have disjoint registers sets such as machines with mostly separate data and address registers that have a few registers in common and machines that have subclasses of registers such as address registers of which some are auto-indexing. Allocation takes place within a basic block boundary. In this phase, the register requirements for each class of registers are simply counted. If more variables than registers of the type needed exist, spill code is added.

During allocation, there may be different possible code combinations that will achieve the proper code but that use different register classes. For example, a variable could be allocated to a general purpose address register that allows all operations to be performed on it or it could be allocated to a subclass of registers that only allows increment and decrement operations. The possible combinations are enumerated, and the code sequence of shortest length is chosen. If spilling is necessary, two possible places can be the destination: either memory or another register class. The decision of where to spill is based on whether there are free registers for a spill to take place into and whether the spill is possible into the free class. Note that spilling into memory always alleviates the overflow problem[1] whereas spilling into a subclass might not; there could still be the problem later of placing too many variables into too few registers. To spill, we need to find a variable with a dormant subtrack that is wholly contained within the live track of another. To spill this variable will reduce the overflow problem by one. Note also that if we spill from a subclass,

[1] we assume sufficient memory for spilling always exists

the spill will relieve the overflow problem in both the subclass and any classes which contain that subclass. Therefore, heuristics are used to spill from subclasses with a higher probability than spilling from more general classes of registers.

3.3 Table Driven

The third allocation and assignment method proposed by P-Y.R. Ma [Ma78] is shown implemented in [ML83]. This method works on the basic block level as with the previous method. The method processes each block is processed to find an initial state that is dependent on the live-out variables from the previous block or blocks. It might be necessary to use additional register-to-register moves (or 'Loads' and 'Stores') to make sure all the preceding final states are compatible with the current initial state. This uncompatibility problem can occur when the compiler has, in different predecessor blocks, assigned the same register to different variables. In this case, for each predecessor block, if the variable's value differs from main memory, it is 'Store'd back into main memory. Then, the register in question is marked as free.

The intermediate code is then macro expanded to the operations needed for the targeted architecture. The machine is assumed to have single function units and a homogeneous register space of limited extent. There may be more than one macro expansion that will produce the desired code in the machine. The decision of which expansion to choose is based on local information for the least-cost instruction sequence. This method uses a cost factor determined by run-time execution length; however, the cost factor could also be based on final program length or some other metric.

After the code for a block has been expanded, the micro-operation's symbolic operands are assigned to registers. This assignment process starts at the top of the block where the final state from the previous block or blocks defines this block's initial state. If a free register exists, it is allocated to the operand. If there are not any free, a register must be de-allocated and the register state updated. The choice of which register to de-allocate is, of course, an important one. The attributes of the variables contained in each register determines which will be spilled. Several attributes are considered in this decision:

- whether the variable is local or a global, with local ones more likely to be spilled because chances are that global ones will be used more often in the succeeding code.
- whether the register contents are active or inactive; that is, whether they differ from the area in main memory that was allocated for the variable. This obviously assumes that the compiler sets aside memory for all variables used in the program, an assumption provided for by an earlier phase of the compiler. If the variable is active, an extra store is necessary to update the memory to reflect the spilling. (This is similar to a dirty bit on a paged-memory operating system. See [**?**] for a discussion of this.)
- whether the variable has been referenced since it has been allocated and whether it will be referenced again.

The following table lists the order that variables will be spilled starting with the ones most likely to be spilled:

1. Passive local variables not used later in the current procedure.
2. Active local variables not used later in the current procedure.
3. Passive global variables not used later in the current procedure.
4. Active global variables not used later in the current procedure.
5. Passive local variables that will be used later in the current procedure.
6. Passive global variables that will be used later in the current procedure.
7. Active local variables that will be used later in the current procedure.
8. Active global variables that will be used later in the current procedure.

When the processing of the current block is done, its final state must be defined by matching the final state of this block with all the initial states of the blocks it flows into. ‘Loads’ and ‘Stores’

might need to be added to insure proper interfacing. One instance when additional 'Loads' and 'Store's may occur is when one or more of the blocks that precede the current block have already been processed. This can happen when branching occurs to some part of the code that has already been expanded and the registers for that code have already been allocated. If a variable exists that is active-out in this block and passive-in in the preceding block, an additional store is needed to update the variable.

Now for a discussion of the graph coloring method.

Chapter 4

THE GRAPH COLORING METHOD

4.1 Introduction

In Chaitan's method ([CAC+81],[Cha82]), register allocation and assignment occur during the same phase. This happens late in the compilation process. It also occurs on the basic block boundaries. As is often the case in computer science, mathematics shows us the way to express and then solve the problem we are faced with. Here the problem is using available machine registers well, given the fact that variables with intersecting live tracks cannot share the same register. If we think of the variables as nodes in a graph and interference between the variables as arcs connecting the nodes, the problem can be stated as trying to find a coloring of the graph that has n colors, where n represents the number of registers in the target architecture. A graph is considered colored if all of a node's neighbors (those connected to it by arcs) can be assigned a color different than the node's color. The terms "neighbor" and "adjacent" convey the same information and will be used interchangeably.

4.2 Building the Interference Graph

As mentioned before, two variables interfere if they are live at the same time. This implies that they must reside in different registers at run time. What needs to be known then, is which variables have intersecting live tracks. A method that would work is: at each program statement, add any defined variables to a list of those already live and remove those that die. While this method works, it is not as efficient as possible because all the required information was already found during data-flow analysis. Keep in mind, the nodes in the graph represent the individual program variables or the compiler-generated variables. Arcs specify for each node which variables

conflict with that node; that is, which variables are live at the same time and therefore cannot reside in the same register. The method adopted for defining the interferences specifies that all variables that are live when a new variable is defined interfere with the new variable. So the process involves finding all the definition points for all the variables and then determining which variables are live at each of those points. This reduces the problem from the line-by-line approach substantially since there are generally few definition points for each variable. And, this information (along with the live tracks for each variable) is known from the data-flow analysis.

A fine point can be observed here concerning the time a variable dies if it corresponds to the definition point of another variable. If the operation has some implicit storage method for the intermediate value, the two variables do not actually interfere–as the live track information might imply. For example, in the statement:

A := B + 3;

if A is born and B dies in this statement, these two variables would not interfere in most architectures because the addition's result would be temporarily stored in the ALU or some multiplexer. It could then be returned to the same register which could now be allocated to A. If the architecture has timing constraints that do not allow writing back into the same register in the same instruction, or if any similar constraint disallows this operation, the variables would interfere, making this point moot.

Some implementations of this graph coloring technique use a method called coalescing to make the graph "easier" to color before actually trying to color the graph. This coalescing process combines nodes that do not interfere into one node, which includes all the arcs from all coalesced nodes. This does not break our notion of interference and does, indeed, make the graph smaller. Coalescing can be beneficial to remove unnecessary register-to-register moves present in the intermediate language. These moves can occur when the source and destination of a register copy operation do not interfere and, therefore, can share the same register. This process of removing the unnecessary register copy operations is known as subsumption. After the coloring process has completed, any coalesced variables are assigned the same register. One drawback to this method

is that it can make some undesirable assignments that could have been avoided had it waited for the coloring process to complete before assigning multiple variables to the same register. This disadvantage occurs because coalescing and subsumption produces strong constraints on final coloring. Some nodes might be coalesced with a already-colored node. Also, there might need to make multiple passes through the graph since coalesces may propagate. Note that this process does not increase the chance that a graph will be n-colorable, only that the determination will be faster because of the resulting smaller size of the graph.

The graph coloring method provides the possibility of 'pre-coloring' sections of the interference graph. This pre-coloring can be used to show programmer-bound register resources. For example, if the programmer binds a variable to a specific register in the source code, the actual register becomes a node in the graph that interferes with all other variables with which its live track overlaps. A more naive approach would remove the bound register from the set of usable colors, but this approach would not take into account the possibility of the binding dying before the end of the basic block. Pre-coloring can also show bindings made by the coloring algorithm in previous blocks. This treatment of previously decided bindings is a consistent and logical extension to the graph coloring procedure.

The concept of interference can be used to express machine idiosyncrasies. Extra arcs can be inserted between nodes known to interfere by some 'feature' of the target architecture for nodes that would not normally be considered interfering by live-track analysis. An example of this extension of the notion of interference is if some machine has a set of registers which are always destroyed on a subroutine call. The variables that are live before and after the call are made to interfere with the destroyed registers. In this way, the variables that are live before and after will not be assigned to the destroyed registers, and, therefore, it is not necessary to do additional saving of those variables before the call. The feature in some architecture where register R0 cannot be used as the base register in an array reference can be expressed by having it interfere with all array variables.

4.3 An Exhaustive Method of Graph Coloring

It is well known that given a graph G and a natural number $n > 2$, the problem of determining whether G is n-colorable is NP-complete, see [Kar72]. Specifically, instances occur of graphs that require time exponential in the size of the graph to decide colorability. The exhaustive method for solving the colorability problem chooses a node to be colored and guesses a color for it. This process continues, and if any guess turns out to be incorrect (that is, does not satisfy the condition of neighbors having different colors), the algorithm backtracks in search of a solution.

An urgency criterion can be used as a greedy method for picking the next node to be colored. Urgency for a node is defined as the current number of uncolored adjacent nodes divided by the remaining number of possible colors. The node with the highest urgency is chosen to be colored next. This urgency criteria can speed the process for coloring graphs that are, in some sense, not difficult to color. These graphs are described as non-pathological. This urgency criteria will, in the worst-case, run in exponential time. However, in most non-pathological graphs it will derive an answer rapidly. Therefore, this algorithm is generally executed only a small number of times, or it is let to run to completion if there is a need or a desire to find a coloring, even in a pathological case. Allowing the completion of the algorithm could be desired when the programmer wants a high degree of optimization. However, to achieve this high degree of optimization, the programmer must pay the penalty of the possible exponential slowdown at compile time to assure having all possible variables in registers at run time. An interesting point is that the slowdown would appear random to the programmer since the addition of or movement of one variable could increase the time required from polynomial time to the exponential realm.

4.4 The Heuristic Approach

A simple observation can be made about graphs, such as those built by variable interference, that make them easier to n-color. This observation is: ***if we remove a node from the graph that has degree less than n, no matter how its neighbors are colored, there will be one color left over for the node.*** The *degree* of a node is the number of neighbors it has. So, for example, if we remove a

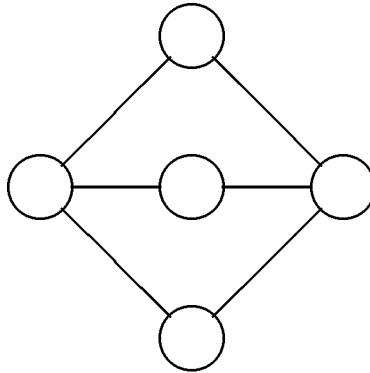

Figure 4.1: A Pathological Graph

node of degree three from a graph we are trying to 4-color, the node has three neighbors that can assume at most three colors, and thus the fourth color will be left for the node. Now, with the new graph that has the node removed, the procedure again searches for a node with a degree less than n. This removal process continues until either no nodes remain in the graph or no nodes in the graph have a degree of less than n. If the graph is empty, there is an n-coloring of the graph, and the procedure is halted. This algorithm needs no backtracking because it is deterministic. That is, the algorithm never makes a mistake when choosing a node to remove because when it removes any node, the degree of the total graph is reduced which can only help to color the graph. If there are no more nodes of degree less than n and the graph is not empty, the graph is not n-colorable using the heuristic method. Therefore, either the exhaustive algorithm can now be run to decide colorability or some variables must be spilled to memory. Note, however, that even if we run the exhaustive algorithm, spilling will probably occur. This is because the heuristic method colors graphs that are non-pathological as well as the exhaustive method and the graphs produced by register interference are usually non-pathological. This heuristic method runs in time that is linear based on the size of the graph. This is obviously much better than the exhaustive method discussed above. The heuristic method will not color graphs considered to be pathological or difficult to color that the exhaustive method outlined above can color. See Figure 4.1 for a 2-colorable graph which cannot be 2-colored by the heuristic approach since it does not have nodes with degree less than 2.

Once a graph has been determined to be n-colorable, assignment of the actual registers generally proceeds by choosing the variables in the inverse order they were thrown out, although order is arbitrary. A check determines what previously register-assigned variables the chosen variable interferes with. This check is performed by observing the neighbors' colors of the node that represents the chosen variable in the variable interference graph. A register (color) unused by any interfering variables is then selected for the chosen variable.

In Chaitan's implementation, two major data structures are used: adjacency vectors and an adjacency matrix. Two structures are needed because, at certain points, a quick check needs to be made to see if two variables interfere. Furthermore, at other points, we need to know which variables interfere with a given variable. The matrix quickly provides the singular interference information. The vector provides a list to traverse when all the interferences of a given variable are needed.

4.5 Spilling

When a graph is not n-colorable, spill code must be introduced to move some variables to memory to free enough registers to proceed. Choosing which variables to spill is an important task that greatly affects the resulting code because of the slowdowns associated with the memory references. Spill code is generally inserted into the intermediate code, the graph is rebuilt, and coloring is attempted again. This process might run several times to obtain a n-colorable graph.

Spilling a variable is not the same as eliminating it from the graph. The variable will still need to be loaded into a register before its value is needed for a computation. Note that all binary operations may be performed with only m machine registers available on a m-address machine. Machines that contain more than the required m registers allow for results from one operation to be the source for succeeding operations without the penalty of accessing memory. Spilling a variable produces a greater number of variables with shorter live tracks which will, in most cases, reduce the number of variables the spilled variable interferes with. This can make it possible for the coloring process to continue because the number of interferences in the graph will be reduced.

A subtle point about the cost of spilling a variable arises if nothing dies between the definition of a local variable and its last use. Spilling this variable would not help the coloring process because no other value could be placed in its register. The variable must be reloaded into a register when needed again, and no other register is freed during its live track so the register cannot be used for another purpose. Therefore, the cost of spilling this variable is set to infinity so that it will not be spilled.

An estimate of the spilling cost each variable is made. Estimates on the probability that a variable will be spilled are based on the following:

- the number of definitions and uses for the variable. The more the variable has, the less reason it should be spilled because each reference would require a Load or Store.
- the nesting depth in program loops. The deeper the nesting, the less reason it should be spilled. Inner loops are executed more frequently than surrounding code. The cost of spilling the variable is, therefore, multiplied by ten for each level to reflect this.

When the coloring algorithm is blocked, it chooses a node from the remaining graph to spill by finding smallest value for the cost of spilling divided by the degree of that node. It is hoped that this will remove the nodes least expensive to spill that also interfere with the most other variables, thereby allowing the graph to be colored when the node is removed.

Obviously, spilling is currently ad hoc. It is ad hoc for the same reason that other algorithms are ad hoc: finding the optimal spilling results is computationally difficult in all cases and intractable in most.

Chapter 5

ALGORITHMS + DATA STRUCTURES

5.1 Introduction

While this paper's goals do not include teaching the reader how to program a working version of the graph coloring algorithm now part of the Horizon system, it is hoped that a quick detour into the actual implementation may help clarify the workings of the algorithm and reduce the time required to implement or modify a graph coloring algorithm.

The Horizon system's coloring procedure works at the program function level. This removes the need to keep spill information between basic blocks, simplifying the algorithm. Spill areas cannot be carried between functions since they are separately compiled and no information can be assumed for a functions' register usage. Observe that performing coloring on a function level has no more information available than the method of performing coloring on a basic block level and transferring information between blocks. So either method produces equally good results. The Horizon system unifies all symbolic register tracks between basic blocks so that all references to a specific variable have the same symbolic number, simplifying building the interference graph.

5.2 Graph Building in the Horizon System

A way to determine when two variables' live tracks interfere is needed to build the interference graph. In the Horizon system, this is easy to do using the data-dependency graph produced by the code selector. This method surpasses attempts to deduce register interference information from the serial code sequences produced by other microcode compilers because there is none of the artificial constraints present in serial code sequences. The only constraints that exist in the DDG are those that actually exist within the control and data flow of the target architecture. Serial sequences of

code have an implied ordering that might have no connection with the data dependencies in the source code or the target architecture. The DDG is also machine-dependent so that each different architecture is expressed in its best possible form, whereas using an unchanging intermediate language inhibits the full expression of differing machine architectural features.

The method for determining interference simply compares all the paths in the DDG from the use nodes of one variable to the defining node of the other variable in question (and vice versa). Paths in the DDG indirectly specify the final ordering of the MOs in the output microprogram. If a path exists between node "a" that is above node "b" in the DDG, then all the MO(s) that "a" produces will be output in the final microprogram before any of the MO(s) that "b" produces. Remember that there may be multiple MOs in a node because of the coalescing of $(0, 0)$ resource timings. The combined paths from the defining node of a variable to all its use nodes define the live track for that variable. Using this knowledge, it can be stated that if paths exist in the DDG from all the use nodes of the first variable to the defining node of the second variable (and vice versa), the two live tracks for the variables will be ordered in the output microprogram and, therefore, will be disjoint. Within each basic block, each variable is compared with all the others for overlap. All live-out variables of a block are considered used at an imaginary last (empty) node of the DDG for the block. All live-in variables are considered defined at an imaginary first (empty) node. This completes the notion of interference in the Horizon system.

A special case to observe is when a variable dies at a node where another is born. If the first variable dies at a node that is the definition for the second variable, the two may not interfere. The method of comparing nodes outlined above would say they interfere because there is an assumed path between a node and itself. Another check is, therefore, added to compare the resource timing between the two operations; if the data-dependent min time is greater than zero, the two live tracks for the variables are said to be disjoint. That is, the MOs that use the two variables may not be placed in the same MI. This allows for the two variables to be assigned to the same register. If the data-dependent min time is zero, the possibility exists that the source register could be overwritten before it is read. Thus, in this case, the variables must be placed in different registers.

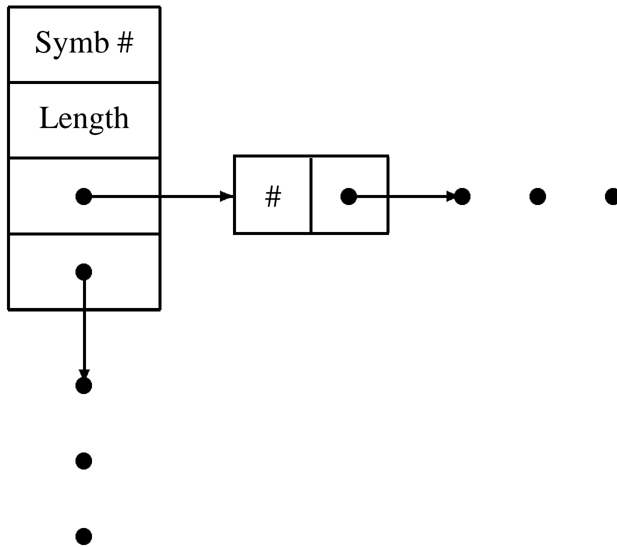

Figure 5.1: Interference Graph Data Structure

5.3 The Implementation of the Graph Coloring Algorithm

The data structure used to represent the interference graph is important since it drives the whole coloring process. Several forms for the overall structure were tried, and the current one appears to have the best combination of size and usability. The data structure used to represent the graph is a linked list of structures which represents the vectors of interference and has length equal to the maximum number of variables in the program in question. Within each structure in the linked list of vectors, there is a pointer to an associated linked list of symbolic variables, and there is an integer ***length*** which represents the length of the associated linked list of symbolic variables. For each symbolic variable in the program, each of which has a structure in the linked list of vectors, the associated linked list points to all other symbolic variables that it interferes with. The variable ***length*** in each structure is, therefore, also the degree of the corresponding node in the interference graph. See Figure 5.1 for a representation of this data structure.

The algorithm to actually color the generated graph puts this structure to good use. Given N hard registers available, the routine proceeds as follows:

1. Start searching in the linked list of vectors for a length which is less than N and greater than "-1".

2. If such a length is found, traverse the associated linked list to find the symbolic register numbers of its adjacent nodes in the graph and decrement the ***length*** value in the data structure for each of these adjacent nodes. Set the ***length*** for the chosen node to "-1" (signifying that it is no longer in the graph). With these two steps, the program removes the node from consideration in the coloring algorithm without actually moving elements or pointers since the algorithm considers only the ***length*** variables. Push the variable number onto a stack maintained to keep track of the variables eliminated from the graph. Start over at step (1) with the new representation of the graph.

3. If all the ***length***s are less than "0", the graph is colored, and we can stop. Pop the entries off the stack and give them hard registers. These are determined by checking all of a variables' neighbors in the graph to determine which hard registers have already been assigned to its neighbors, and then choosing a register that is still free.

4. If all nodes that have ***length*** greater than "0" also have ***length*** greater than or equal to N, the coloring process cannot continue. Spill code needs to added to remove one or more nodes so that the coloring process can continue. The code needed to perform spilling is added at the ssm level. The resulting intermediate code is run back through the code selector and the coloring algorithm is rerun on the modified code. This is repeated as many times as necessary (generally only once) to produce an N-colorable graph.

An actual example with the steps shown with the various stages of data structures might further illustrate this process. Let us assume that there is some machine with only three registers available for any program variables. The goal then is to color an interference graph for a program with only three colors ($N = 3$). Given the basic block in Figure 5.2, assume the front-end of the compiler has assigned the number "1" as the symbolic register for the variable "a", "2" for "b", "3" for "c" and "4" for "d". Figure 5.3 represents the interference graph for this basic block. The initial data structure used to represent this graph is shown in Figure 5.4.

The algorithm would then traverse this data structure looking for a ***length*** of less than N. The first ***length*** of less than N it finds is at symbolic register "2" which has ***length*** equal to 2.

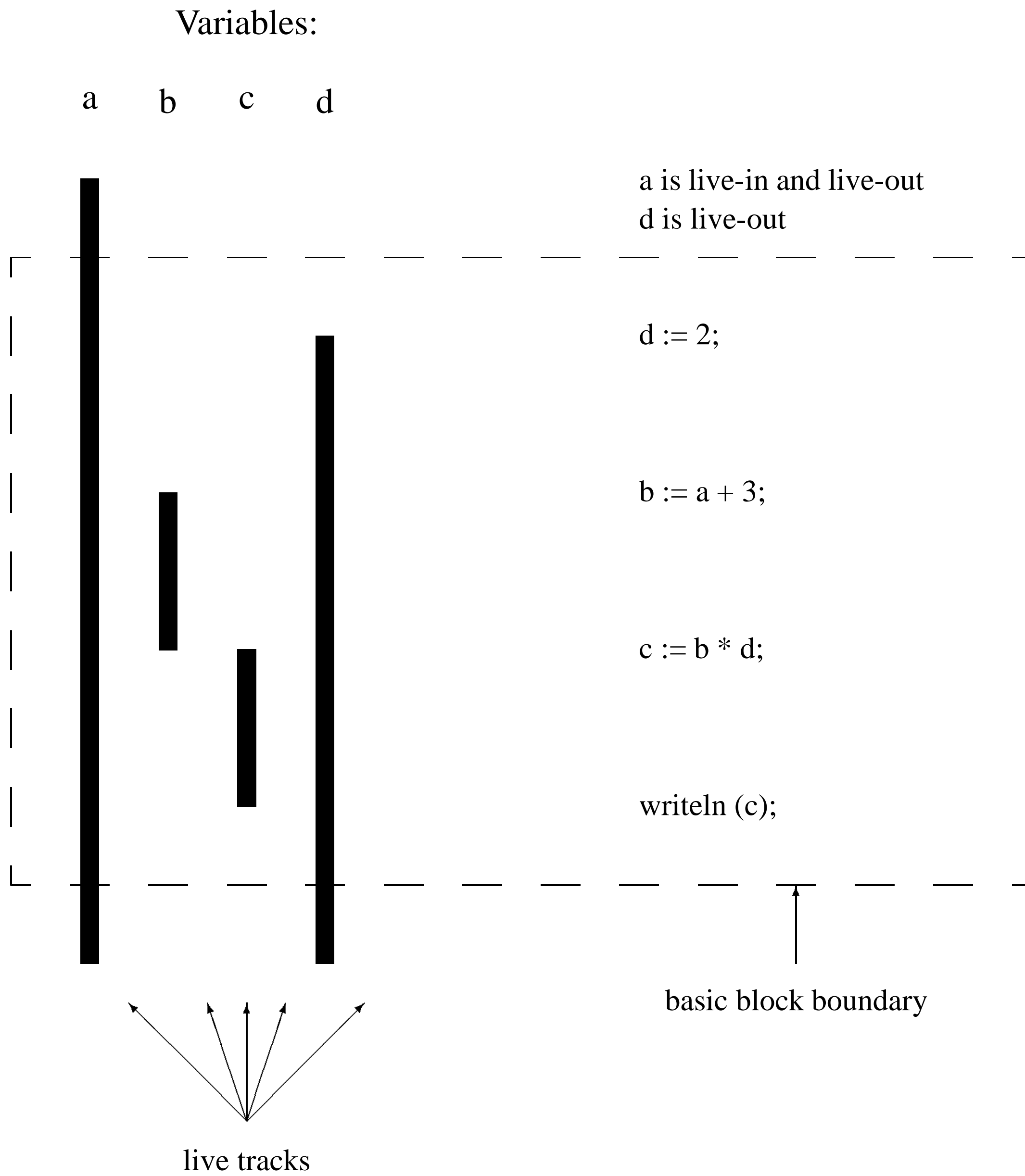

Figure 5.2: An Example Basic Block Showing Live Tracks

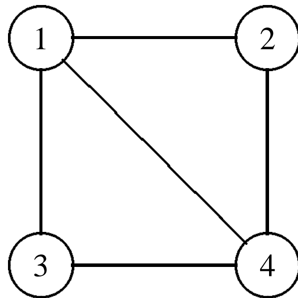

Figure 5.3: The Interference Graph for the Example

The algorithm then sets the variable ***length*** for this index equal to “-1” to denote that this node no longer exists in the modified graph. There is no reason to actually move array members or pointers around since the coloring algorithm relies only on the ***length*** variables. The linked list associated with symbolic register “2” is then traversed, and all numbers encountered represent the symbolic registers for the adjacent nodes in the interference graph. Each of these numbers is used to decrement the proper ***length*** variable in the associated symbolic registers’ structure in the linked list of vectors. See Figure 5.5 for the new representation of the graph.

The scan of the linked list of vectors is repeated, and the first element is found to have ***length*** of less than N. The first node is therefore “thrown out” by making its ***length*** “-1” and the adjacent nodes in the interference graph are decremented as before. Figure 5.6 shows the data structure after two iterations of the algorithm.

Two iterations are still needed to color the graph; however, they should be self-evident. The first of the remaining iterations would remove the node at symbolic register “3”. The second would remove the node at “4”, and all the ***length***s would then be less than zero showing the graph is 3-colorable.

The variables represented by the nodes are then assigned to hard registers in the reverse order of when they were thrown out. Then, a check is made to see what colors (registers) a nodes’ (variables’) neighbors have been assigned, and a color is chosen that is still free. If the coloring

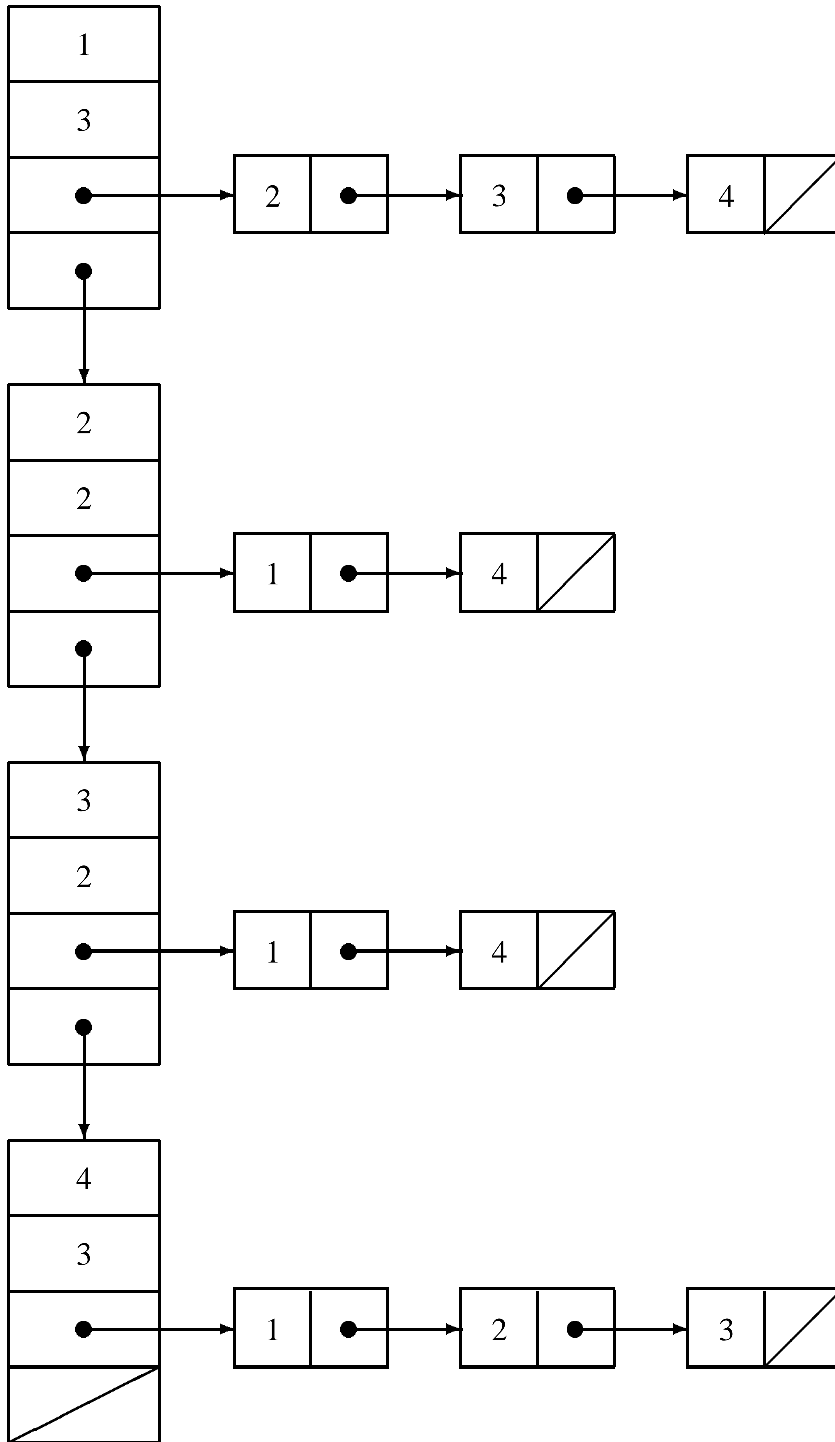

Figure 5.4: The Initial Interference Graph Data Structure for the Example

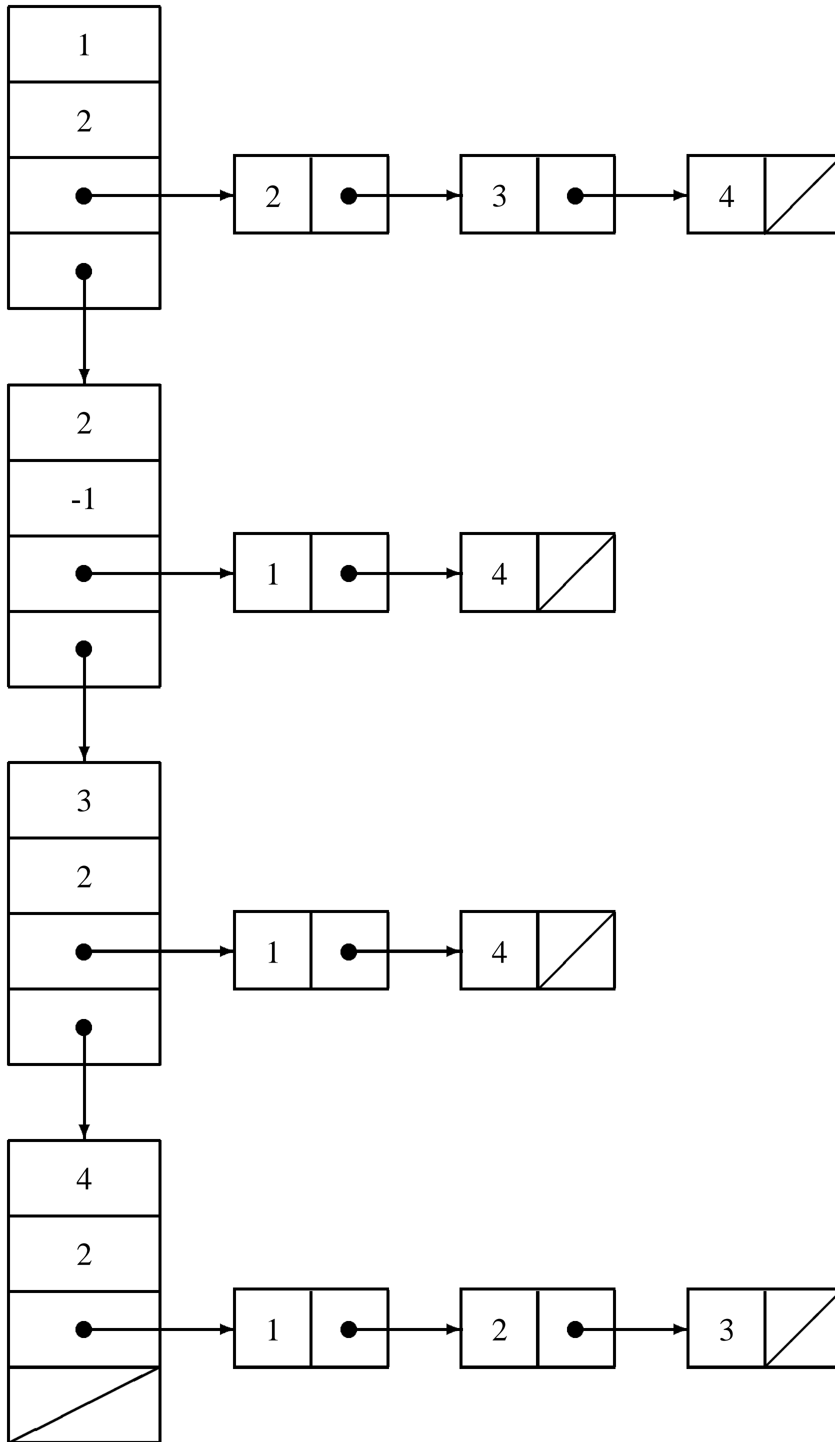

Figure 5.5: The Graph Data Structure after One Iteration

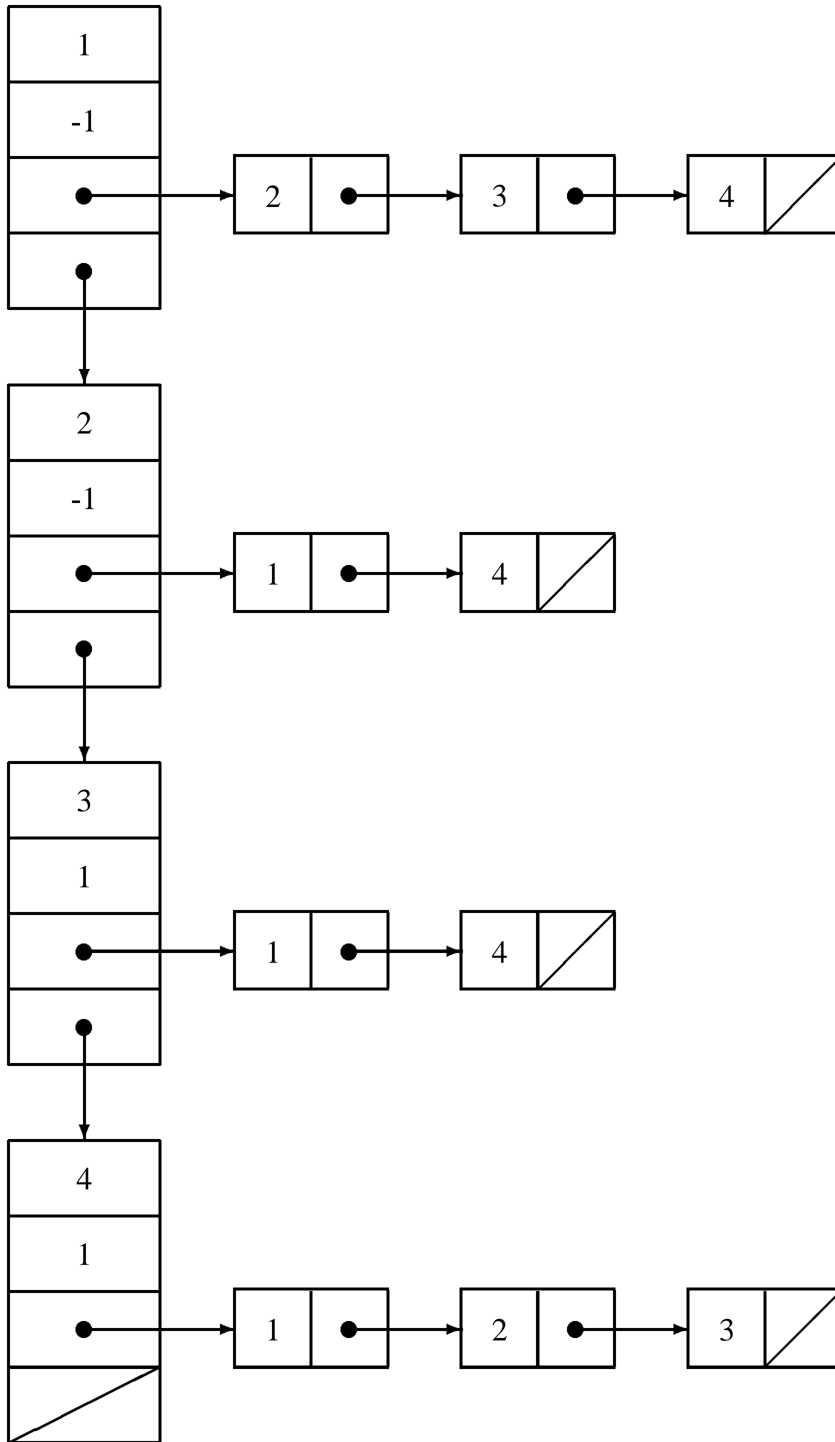

Figure 5.6: The Graph Data Structure after Two Iterations

algorithm fails, as it would have if it had tried to find a 2-coloring for this graph, a heuristic spilling procedure is invoked.

5.4 Spilling

When the coloring algorithm is blocked, a number of variables must be removed by spilling them to some secondary storage area. To determine which of the possible variables is to be spilled, an estimate of spilling cost is needed. The current version uses the total number of use points (since there is only one definition point per track) divided by the degree of the node as an estimate of spilling cost. Nesting level is taken into account by multiplying the estimated cost by an approximation of the number of times a loop is executed; the current approximation is ten times. All definition and use points are considered when computing the additional cost for nesting level. This method of estimation, hopefully, removes nodes used the least while at the same time effectively unblocking the coloring process. Programmer-defined bindings receive large, but not infinite, spilling costs. This guarantees that they will reside in machine registers in all but the most pathological situations. If they are given infinite spilling cost, it would be possible to completely block the register allocation process. For example, if the programmer binds too many machine registers, insufficient registers would be free for basic computational needs.

Any local variable that has a live track that does not contain final use nodes for another variable has its spill cost estimate set to infinity. It cannot help the colorability of the graph if that local variable is spilled because a register will be required for the variable to perform the needed operation, and the variable would have to be reloaded into a free register. Yet, none will be free since no other variables die within its live track. This method also keeps the spilling algorithm from attempting to spill a previously spilled variable. The need to disallow spilling of these types of variables arises from the fact that most micro-architectures do not allow operations to be performed with operands from secondary memory. Machines that do allow operands to reside in secondary memory simplify register allocation and assignment by removing the need to place spilled variables back into the available register banks.

Instead of spilling a variable, it may be possible to recompute its value when the value is needed again. It is possible this re-computation could be done at no cost, depending on the availability of empty MO's in the MI's preceding the MI where the value is needed. Unfortunately, this situation is very difficult to recognize in general, so it has little practical value. A more likely situation requiring no additional computation occurs when spilling does not add to the length of the produced microcode. This is possible when the code required for the spill fills empty MO's in the MI's where the spill code is placed. The local compaction algorithm recognizes this situation automatically.

It is hoped that this minor detour into the workings of a graph coloring algorithm has made it easier to see the underlying process.

Chapter 6

RESULTS

6.1 Introduction

In this chapter, results of the graph coloring algorithm currently in place in the Horizon compiler are shown.

6.2 Example #1

Example #1 was contrived to show a simple basic block that is 4-colorable. It also shows how programmer-bound variables can be intermixed with unbound variables without inhibiting assigning bound registers to unbound variables. There are no live-in and live-outs in this example. Figure 6.1 shows the Micro-C code used to produce this basic block. Only assignment of constants and variables are used to simplify the demonstration. Live tracks for the various variables are shown in the comments following the source statements. The interference graph for this block produced as an integral part of the algorithm is shown in Figure 6.2. The output produced by the graph coloring algorithm is shown in Figure 6.3. Included is the adjacency vector output, the color stack and the actual assignment of hard registers to variables. The “sym” column of the output shows the choice of symbolic register numbers made by the front-end of the compiler. As can be seen from the assignment of hard registers, symbolic registers are represented internally as the program variable name concatenated with a unique number. Different live tracks for a variable receive different numbers to differentiate them. The name of the variable concatenated with the machine register associated with that variable denotes programmer-bound variables. Note that both machine registers #0 and #1 are assigned to variables falling outside of the live tracks of those programmer-bound variables. This shows that programmer bindings are limited to their actual def-use chains, and those registers can be used for other variables in a basic block.

```
void test()
{
reg w,x,y,z;
r0 a;
r1 b;
r2 c;

                /*      live tracks      */
                /*      ---- ------      */
                /*  a  b  c  w  x  y  z  */
      c = 1;    /*        .              */
      x = 10;   /*        |     .        */
      z = 20;   /*        |     |     .  */
      w = 30;   /*        |  .  |     |  */
      y = z;    /*        |  |  |  .  |  */
      a = y;    /*  .     |  |  |  |     */
      b = w;    /*     .  |  |  |        */
      b = c;    /*     .  |     |        */
      a = x;    /*  .           |        */
      b = a;    /*  |  .                 */
}
```

Figure 6.1: Example Program #1

6.3 Example #2

Example #2 was contrived to show that coloring takes place across basic block boundaries within a function. The Micro-C code is shown in Figure 6.4. In this example, the variable “a” is live-through both clauses of the “if-else”. It therefore must interfere with both variables “b” and “c”. Variable “b” does not interfere with variable “c” because the two sections of code within the “if-else” structure cannot be executed at the same time and therefore “b” and “c” may be assigned the same register. See Figure 6.5 for the output of the graph coloring algorithm.

6.4 Example #3

This example shows spilling cost estimates. The program in Figure 6.6 produces a 4-colorable interference graph but the number of hard registers for this example is set to 2. This creates a need to spill two variables. The list of costs show that the following heuristics are used:

1. Each use adds one to the cost.

```
The current adjacency vectors:
 var  sym  degree  adjacents
 ---  ---  ------  ---------
  r2    2    4       y141    w140    z139    x138
   x  138    4       y141    w140    z139   r22
   z  139    3       w140    x138   r22
   w  140    4       y141    z139    x138   r22
   y  141    3       w140    x138   r22
  r0    0    0
  r1    1    0

was able to color graph

The current color stack is :
  r11     r00      y141    w140    x138   r22      z139

The assignment:
 var  register
 ---  --------
  r2     2
   x     3
   z     0
   w     1
   y     0
  r0     0
  r1     1
```

Figure 6.2: Output of Coloring Algorithm for Example Program #1

```
void test()
{
reg a,b,c,d;

   a = 10;
   if (true(1))
   {
      b = 20;
   }
   else
   {
      c = 30;
   }
   d = a;
}
```

Figure 6.3: Example Program #2

```
The current adjacency vectors:
 var  sym  degree  adjacents
 ---  ---  ------  ---------
   c  140    1       a143
   d  142    0
   b  139    1       a143
   a  143    2       c140    b139

was able to color graph

The current color stack is :
   a143    b139    d142    c140

The assignment:
 var  register
 ---  --------
   c     1
   d     0
   b     1
   a     0
```

Figure 6.4: Output of Coloring Algorithm for Example Program #2

```
void test()
{
   reg a,b,c,junk;
   r1 reg1;

   a = 10;
   reg1 = 20;
   while (true(1))
   {
      b = 30;
      while (true(1))
      {
         c = reg1;
      }
   }
   junk = b;
   junk = a;
   junk = reg1;
   junk = c;
   junk = a;
}
```

Figure 6.5: Example Program #3

2. Each use within a deeper nesting level multiplies the cost by ten.

3. Programmer-bound variables have an additional cost of 100000.

4. The cost computed so far is divided by the degree of the node.

The value 100000 is added for programmer-bound variables to assure that variable will remain in a register in most cases. An unbound variable with a nesting depth of greater than five can cause a non-nested programmer-bound variable to be spilled.[1] Figure 6.7 shows the cost estimates and the variables chosen to be spilled. A value for the spilling cost of "-1" is taken to be infinite. The "junk" variables are used only to complete the live tracks for other variables. Since they have no use nodes, there is no reason to spill them so their spilling cost is set to infinity.

[1]This probably gives the programmer too much credit for the choice of bindings. If a case arises when the algorithm would choose to spill a programmer-bound variable, a warning could be issued to inform the programmer of the bad choice of binding.

```
The current adjacency vectors:
 var  sym  degree  adjacents
 ---  ---  ------  ---------
junk  142    0
junk  144    0
junk  145    0
junk  147    0
junk  148    0
   c  149    3       b150    a151   r11
   b  150    3       c149    a151   r11
   a  151    3       c149    b150   r11
  r1    1    3       c149    b150    a151

wasn't able to color graph

       var       cost
       ---       ----
      junk142    -1.000000
      junk144    -1.000000
      junk145    -1.000000
      junk147    -1.000000
      junk148    -1.000000
         c149    66.666664
         b150    6.666667
         a151    1.000000
        r11      33433.332031

choose to spill a151
choose to spill b150
```

Figure 6.6: Output of Coloring Algorithm for Example Program #1

Chapter 7

SUMMARY AND DIRECTION FOR FUTURE WORK

In this thesis, several methods of register allocation and assignment have been described. The first was the branch and bound technique, an early method that inherently is computationally expensive. Then, it reviewed the method for non-homogeneous register banks that can spill into registers or memory. Next, the more effective and less complex table-driven method was discussed. Finally, the graph coloring method and its implementation on the Horizon compiler were discussed in detail. It should be pointed out that these methods are answers to the specific questions that arise in various architectures, and none is suitable as a generic method for all machines.

7.1 Extensions to the Graph Coloring Approach

The graph coloring method works well for machines with homogeneous register banks. It has not been used for those architectures with subclasses of registers, although it could be extended to include this type of machine. The extension would be to color from the most specific type of register class, spilling to registers of a more general type. It could then move on to progressively more general types of registers, coloring both variables that need to be in that type of register and those spilled from subclasses. Another type of architecture that coloring might be extended to is the overlapping register windows prevalent in the current RISC (reduced instruction set computer) machines. The method would color all of the registers needed for a procedure and all the registers used to pass the parameters to a called procedure. The called routine would use the assignment made by the calling procedure.

An ultimate extension to the graph coloring method would be to do all of the variable assignments needed by an input program using the graph coloring technique. Since all program

variables are assigned to some type of register-like resource, such as main memory which can be considered to be a large bank of registers, this extension would not be difficult. First, any basic data registers, subclasses first as above, would be assigned. Then the other machine registers such as on-chip pointer files, would be assigned. Next, the data areas in main memory would be colored and assigned. In this way, a very efficient usage of all memory resources could be made. Main memory requirements would shrink because extra space would not need to be set aside for variables whose live tracks do not intersect. Activation stacks would also be reduced in size. Any spilled variables from one level would simply become part of the graph to be colored at the next level in the memory hierarchy. Arrays would have to be treated as singular variables of some fixed length unless memory disambiguation could be performed to identify disjoint portions of them within the code.

An obvious assumption is that enough room is available at some level to hold all the interfering program variables. If this assumption does not hold, no method can assign the program variables to the available machine resources, and the program simply would be unable to fit in the given machine. However, using graph coloring for all register-type resources in an architecture would produce a much better usage of memory than methods that allocate space for all global variables. Most global variables probably would not interfere and could, therefore, use the same memory addresses. The penalty paid for the global graph coloring method is a time increase at compile time, which would increase with the number of program variables present. A foreseeable difficulty with this method is that different memories might need different spilling heuristics. This problem is not difficult to overcome by having the compiler know which type of memory is currently being colored and by having a different set of heuristic routines for each. The consistency in the method of assignment and the resulting space and execution efficiency of the code produced in this manner could outweigh this problem.

7.2 Ease of Retargetability

In the Horizon compiler, retargeting of the actual graph coloring process is trivial. The number of hard registers available in each bank to be assigned are the only variables in this process.

However, spilling is another issue entirely. Each different target architecture may require different heuristics to achieve the best results from spilling. Different types of register banks, sizes of register banks and penalties for spilling to a particular area exist in each machine. A library of spilling heuristics could, though, be built up from a variety of machines. The compiler could have a method to specify the weights for the probability of using each heuristic in a particular machine. These weights would have to be manually adjusted for each new architecture or the compiler would have to learn which set of weights is best for each machine by repeatedly adjusting the weights and observing the compacted results, looking for the most efficient code.

REFERENCES

[All86] V.H. Allan. *A Critical Analysis of the Global Optimization Problem for Horizontal Microcode*. PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1986.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, MA, 1977.

[BDMW86] S.J. Beaty, M.R. Duda, R.A. Mueller, and J.S. Walicki. Horizon code generator table-driven code selector (version 3.1): Prolog implementation. Technical Report MAD-86-7, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, September 1986.

[CAC+81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, and P.W. Hopkins, M.E.and Markstein. Register allocation via coloring. *Computer Languages*, 6, 1981.

[Cha82] G.J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 201–207, June 1982.

[Das84] S. Dasgupta. A model of clocked micro-architectures for firmware engineering and design automation applications. In *Proceedings of the 17th Microprogramming Workshop (MICRO-17)*, pages 298–308, New Orleans, LA, November 1984.

[DeW76] D.J. DeWitt. *A Machine-Independent Approach to the Production of Optimal Horizontal Microcode*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI, 1976.

[DMa] M.R. Duda and R.A. Mueller. ***μ-c. Technical Report CS-84-12, Computer Science Department, Colorado State University, Fort Collins, Colorado.***

[DMb] M.R. Duda and R.A. Mueller. μ-c. Technical Report CS-85-11, Department of Computer Science, Colorado State University, Fort Collins.

[DM84] M.R. Duda and R.A. Mueller. μ-c microprogramming language specification. Technical Report CS-84-11, Department of Computer Science, Colorado State University, Fort Collins, October 1984.

[Ell86] J. R. Ellis. Bulldog: A Compiler for VLIW Architectures***. The MIT Press, Cambridge, MA, 1986. PhD thesis, Yale, 1984.***

[Fis81] *J.A. Fisher. Trace scheduling: A technique for global microcode compaction.* IEEE Transactions on Computers*, C-30(7):478–490, July 1981.*

[GS82] *R. Grishman and B. Su. A preliminary evaluation of trace scheduling for global microcode compaction. Technical Report 042, Courant Institute, New York University, New York, May 1982.*

[GS83] *R. Grishman and B. Su. A preliminary evaluation of trace scheduling for global microcode compaction.* IEEE Transactions on Computers*, C-32(12):1191–1194, Dec 1983.*

[Kar72] *R.M. Karp.* Reducibility among Combinatorial Problems*, pages 85–103. Plenum Press, New York, NY, 1972.*

[KT79] *J. Kim and C.J. Tan. Register assignment algorithms for optimizing micro-code compilers - part i. Technical Report RC 7639, IBM, Computer Sciences Department, IBM T.J. Watson Research Center, Yorktown Heights, NY, May 1979.*

[Ma78] *P-Y.R. Ma.* Optimizing Microcode Produced from a High-Level Language*. PhD thesis, Department of Computer Science, Oregon State University, Corvallis, OR, August 1978.*

[MDSW86] *R.A. Mueller, M.R. Duda, P.H. Sweany, and J.S. Walicki. Horizon: A retargetable compiler for horizontal micro-architectures. Technical Report MAD-86-23, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, December 1986.*

[ML83] *P-Y.R. Ma and T. Lewis. The design of a resource allocation scheme for microcode generation.* Euromicro Journal of Microprocessing and Microprogramming*, 11(5):277–286, 1983.*

[MS86] *R.A. Mueller and P.H. Sweany. Horizon code generator series-parallel ddg coupler/decoupler (version 3.1). Technical Report MAD-86-10, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, September 1986.*

[MV88] *R.A. Mueller and J. Varghese. Fundamental concepts in microprogramming. In S. Habib and S. Dasgupta, editors,* Principles of Microprogramming and Firmware Engineering*. Van Nostrand, New York, NY, 1988.*

[MW86] *R.A. Mueller and J.S. Walicki. Horizon code generator micro-architecture specification and representation. Technical Report MAD-86-13, Firmware Engineering and Micro-Architecture Design Laboratory, Colorado State University, Fort Collins, CO, September 1986.*

[Plo87] *B.L. Plomondon. Targeting the horizon compiler to a signal processor: A case study. Master's thesis, Computer Science Department, Colorado State University, Fort Collins, CO, 1987.*

[SDJ84] ***B. Su, S. Ding, and L. Jin. An improvement of trace scheduling for global microcode compaction. In*** Proceedings of the 17th Microprogramming Workshop (MICRO-17)***, pages 78–85, New Orleans, LA, Nov 1984.***

[SDX86] ***B. Su, S. Ding, and J. Xia. Urpr – an extension of urcr for software pipelining. In*** Proceedings of the 19th Microprogramming Workshop (MICRO-19)***, pages 94–103, New York, NY, December 1986.***

[Set75] ***R. Sethi. Complete register allocation problems.*** SIAM Journal of Computing***, 4(3):226–248, 1975.***

[Tre82] ***N. Tredennick. Cultures of microprogramming. In*** Proceedings of the 15th Microprogramming Workshop (MICRO-15)***, pages 79–83, Palo Alto, CA, Oct 1982.***

[Veg82] ***S.R. Vegdahl.*** Local Code Generation and Compaction in Optimizing Microcode Compilers***. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1982.***