

Local Register Allocation Spanning Basic Blocks

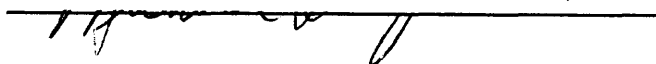
by

Lynn Thompson
B.S., Portland State University, 1983

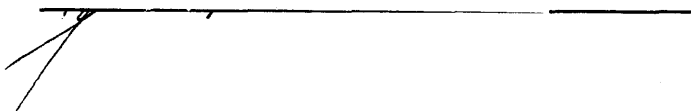
A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering
April 1996

The thesis "Register Allocation Spanning Basic Blocks" by Lynn Thompson has been examined and approved by the following Examination Committee:

Michael Wolfe, thesis advisor
Professor

A horizontal line with a handwritten signature in cursive script above it, positioned to the right of the name Michael Wolfe.

James Hook
Associate Professor

A horizontal line with a handwritten signature in cursive script above it, positioned to the right of the name James Hook.

James Reinders
Microcomputer Software Labs, Intel Corp.

A horizontal line with a handwritten signature in cursive script above it, positioned to the right of the name James Reinders.

Acknowledgements

I thank the faculty of OGI for the opportunity to study at this institution and Intel Corporation for their support of continuing education.

Table of Contents

ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
ABSTRACT	ix
CHAPTER 1 - REGISTER ALLOCATION	1
DISCUSSION OF PROBLEM.....	1
TERMINOLOGY.....	1
STRUCTURE OF THESIS.....	4
CHAPTER 2 - RELATED WORK	5
INTRODUCTION.....	5
OVERVIEW OF GRAPH THEORY.....	6
GRAPH COLORING ALGORITHM.....	9
OVERVIEW OF INTERVAL & CIRCULAR-ARC GRAPHS.....	11
CYCLIC PERIOD GRAPH COLORING ALGORITHM.....	14
HIERARCHICAL PERIOD GRAPH ALGORITHM.....	16
LIVE RANGE SPLITTING.....	17
LIVE RANGE SPLITTING ALGORITHM.....	18
MULTIPLE REGISTER OPERANDS.....	19
FLOW GRAPH TECHNIQUES.....	22
CHAPTER 3 - COMPILER FRAMEWORK	25
DATA FLOW ANALYSIS.....	25
SPANNING TREES IN DIRECTED GRAPHS.....	27
CONTROL FLOW CONSTRUCTS.....	28
LIVE VARIABLE ANALYSIS.....	29
REGISTER ASSIGNMENT.....	32
CHAPTER 4 - LOCAL REGISTER ALLOCATION	34
BACKWARD PASS ALGORITHM.....	34
FORWARD PASS ALGORITHM.....	36
CONCLUSION.....	39
CHAPTER 5 - REGISTER ALLOCATION ACROSS BASIC BLOCKS	41
SUPERTRACE FORMATION.....	41
SUPERTRACE LOCAL VARIABLES.....	43
FORWARD PASS SUPERTRACE ALGORITHM.....	43
OPTIMALITY CRITERIA.....	50
VARIABLE RENAMING IN MODIFIED SUPERTRACE ALGORITHM.....	51
MODIFIED SUPERTRACE ALGORITHM.....	56
CONCLUSION.....	61
CHAPTER 6 - CREATING LARGER SUPERTRACES	62

INTRODUCTION	62
LIMITS	63
BASIC BLOCK CLONING	64
LOOP UNROLLING	72
CONCLUSION	73
CHAPTER 7 - FUTURE WORK AND CONCLUSIONS	74
CONCLUSION	76
BIBLIOGRAPHY	77
BIOGRAPHICAL INFORMATION.....	79

List of Tables

CHAPTER 3

LIVE VARIABLE ANALYSIS INITIALIZATION	32
LIVE VARIABLE ANALYSIS CONVERGENCE	32

CHAPTER 4

BACKWARD PASS BASIC BLOCK REGISTER ALLOCATION	36
BACKWARD PASS BASIC BLOCK REGISTER ALLOCATION RESULTS	36
FORWARD PASS BASIC BLOCK REGISTER ALLOCATION INITIALIZATION	38
FORWARD PASS BASIC BLOCK REGISTER ALLOCATION REGISTER ASSIGNMENT	39
FORWARD PASS BASIC BLOCK REGISTER ALLOCATION RESULTS	39

CHAPTER 5

FORWARD PASS SUPERTRACE REGISTER ALLOCATION INITIALIZATION	49
FORWARD PASS SUPERTRACE REGISTER ASSIGNMENT	49
FORWARD PASS SUPERTRACE REGISTER ALLOCATION RESULTS	50
MODIFIED SUPERTRACE REGISTER ALLOCATION INITIALIZATION	59
MODIFIED SUPERTRACE REGISTER ASSIGNMENT	60
MODIFIED SUPERTRACE REGISTER ALLOCATION RESULTS	60

List of Figures

CHAPTER 1

COMPILER PHASES	2
-----------------------	---

CHAPTER 2

REGISTER LIFETIMES	5
GRAPH OPERATIONS	7
LINE SEGMENTS AND INTERVAL GRAPH	12
PERIOD GRAPHS	13
CYCLIC PERIOD GRAPH AND CIRCULAR ARC GRAPH	14
HIERARCHICAL PERIOD GRAPH	17
GRAPH OF TWO REGISTER TRIPLES AND A REGISTER PAIR	21
GRAPH WITHOUT IMPLICIT EDGES	21
NORMALIZED GRAPH WITHOUT IMPLICIT EDGES	22

CHAPTER 3

ANNOTATED FLOW GRAPH	26
DEPTH FIRST SEARCH ALGORITHM	27
SPANNING TREE NUMBERING	29
LIVE VARIABLE ANALYSIS ALGORITHM	31

CHAPTER 4

BACKWARD PASS BASIC BLOCK REGISTER ALLOCATION ALGORITHM	35
FORWARD PASS BASIC BLOCK REGISTER ALLOCATION ALGORITHM	37

CHAPTER 5

SUPERTRACE	42
DRIVER ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION	44
INITIALIZATION ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION	45
ALLOCATION ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION	46
STATE SAVING ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION	47
EXAMPLE FLOW GRAPH FOR SUPERTRACE REGISTER ALLOCATION	48
DRIVER ALGORITHM FOR VARIABLE RENAMING	52
BACKWARD TRAVERSAL ALGORITHM FOR VARIABLE RENAMING	53
FORWARD TRAVERSAL ALGORITHM FOR VARIABLE RENAMING	54
EXAMPLE FLOW GRAPH BEFORE VARIABLE RENAMING	55
EXAMPLE FLOW GRAPH AFTER VARIABLE RENAMING	55
DRIVER ALGORITHM FOR MODIFIED SUPERTRACE REGISTER ALLOCATION	56
INITIALIZATION ALGORITHM FOR MODIFIED SUPERTRACE REGISTER ALLOCATION	57
ALLOCATION ALGORITHM FOR MODIFIED SUPERTRACE REGISTER ALLOCATION	58

CHAPTER 6

DRIVER ALGORITHM FOR BASIC BLOCK CLONING	66
TRAVERSAL/GENERATION ALGORITHM FOR BASIC BLOCK CLONING	67

EXAMPLE FLOW GRAPH BEFORE BASIC BLOCK CLONING	68
EXAMPLE FLOW GRAPH DURING BASIC BLOCK CLONING	69
EXAMPLE FLOW GRAPH DURING BASIC BLOCK CLONING	70
EXAMPLE FLOW GRAPH AFTER BASIC BLOCK CLONING	71
LOOP UNROLLING ALGORITHM	73

Abstract

This thesis addresses the problem of how to perform local register allocation on variable live ranges that span basic block boundaries. Register allocation is often performed in two passes. Local register allocation applies to variables with live ranges contained in single basic blocks. Global register allocation assigns registers to variables with live ranges that cross basic block boundaries. We propose to extend the applicability of local register allocation by using a fast algorithm to allocate registers whose lifetimes span basic blocks but are local to control regions that we call a supertrace. We present basic block cloning and loop unrolling as optimizations to increase the size of supertraces. Two basic block register allocation algorithms and two supertrace register allocation algorithms are presented in detail.

Our goal in this research is to provide evidence that supertrace optimization is a viable method of increasing the number of register references processed by the local register allocation routine.

CHAPTER 1

REGISTER ALLOCATION

Discussion of Problem

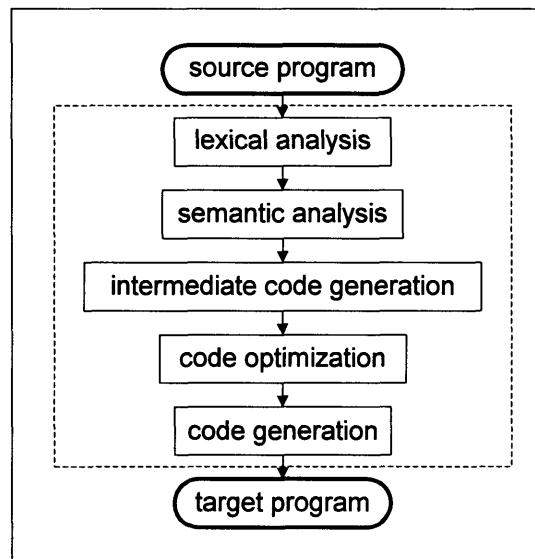
Basic block local register allocation has the shortcoming that variables eligible for allocation must have live ranges wholly contained within the basic block. This thesis addresses the problem of how to extend the live ranges of variables beyond basic block boundaries by identifying single entry, multiple exit regions in a control flow graph. Performing local register allocation using the granularity of a region or *supertrace* can potentially increase the number of variables eligible for local register allocation.

This thesis presents algorithms for local register allocation demonstrating the concepts and functions of local register allocation within basic blocks and supertraces. We also present algorithms to create larger supertraces by unrolling loops cloning basic blocks. Analysis of the cost/benefit trade-offs support future implementation.

Terminology

In this chapter we introduce terms and concepts for the register allocation phase in a typical compiler. Register allocation determines which variables are stored in physical registers and which variables are referenced from memory. Heuristics drive the register allocation process, taking into account the number of registers available on a given architecture, the usage patterns for variables, and a number of cost/benefit trade offs. We present the three heuristics for register allocation algorithms, noting the

benefits and shortcomings of each. Figure 1-1 illustrates the phases in a 'common' language compiler that converts a source program file into a target program file.



COMPILER PHASES
FIGURE 1-1

In the sequence of compiler phases required to convert a source program into a target program, register allocation happens before code generation. Because we use a flow graph representation, information about variable usage and flow control structures is available to the register allocation algorithms. It is beyond the scope of this research to explain the operation of each phase in detail; we refer the reader to a compiler textbook[1].

The input to the register allocation phase is a flow graph constructed from a subroutine or function. Groups of instructions in the flow graph create *basic blocks*. A basic block has a single entry and exit point. Instructions or operations contained in a basic block have no external entry points or branches out of the basic block to interfere with the sequential execution of every instruction. Register allocation algorithms replace variable references with register references. Register allocation is often a two pass operation. The local register allocation pass deals with variables whose live range is within a basic block. Global register allocation applies to variables with live ranges spanning several basic blocks. *Live range analysis* (presented in Chapter 3) determines whether the live range for a variable spans several basic blocks.

Register allocation heuristics must consider the number and type of physical registers, instruction set limitations for operands, the memory hierarchy, and the efficiency of accessing a value within the memory hierarchy. All register allocators must provide a mechanism to temporarily free an assigned register by storing the value in memory and then retrieve the value before its next use.

Assigning all virtual registers to physical registers is an aggressive approach to register allocation. Values stored in physical registers alleviate the need for memory stores and fetches of instruction operands. The benefit of immediate access to values stored in a register cannot be realized without a sufficient pool of general purpose registers. The hardware architecture may dictate the use of this register allocation policy, as in arithmetic operations on load/store architectures. Many Reduced Instruction Set Computer (RISC) machines are load/store architectures.

Storing loop indices in physical registers is another approach to register allocation. Accessing memory for variables references in the loop is costly. Secondary heuristics can optimize or refine the allocation registers based on other criteria. Secondary heuristics might include allocation based on the length of the lifetime for the virtual register.

Keeping all virtual registers in memory is costly in terms of memory access during program execution. Implementing this strategy in a compiler would cause delays or *bottlenecks* due to insufficient memory bandwidth between the processor and memory. This policy does not provide the benefit of long register lifetimes since every virtual register has a life span of one instruction. Reading each operand from memory before execution and storing it after execution would increase execution time for generated code. Perhaps the only benefit to this model is the freedom from register pressure caused by overlapping live ranges. Different processor architectures may or may not support this model for register allocation, making portability a concern.

Structure of Thesis

In Chapter two we present information from graph theory pertinent to our research. We use information from general graph theory as a basis to discuss interval graphs, circular arc graphs, and the graph coloring problem. We address the similarities and differences between several approaches to register allocation. We relate previous research to our efforts.

Chapter three provides detailed information and algorithms for spanning trees and live variable analysis. Algorithms presented in this chapter analyze the flow graph and variable usage patterns for the register allocation process.

We present two basic block register allocation algorithms in Chapter four. The basic block register allocation algorithms use live variable analysis to identify basic block local variables. The register allocation algorithms assign registers for basic block local variables. We discuss the benefits and shortcomings of this approach to local register allocation and the behavior of the algorithms when there are too few physical registers to hold all simultaneously live variables in the basic blocks.

In Chapter five we present our main result, two register allocation algorithms that permit local register allocation of variables that span basic block boundaries. We find single entry, multiple exit regions in the flow graph called *supertraces*. We present the algorithms and examples of local register allocation within supertraces. We discuss the benefits and shortcomings of these algorithms, and study their optimality.

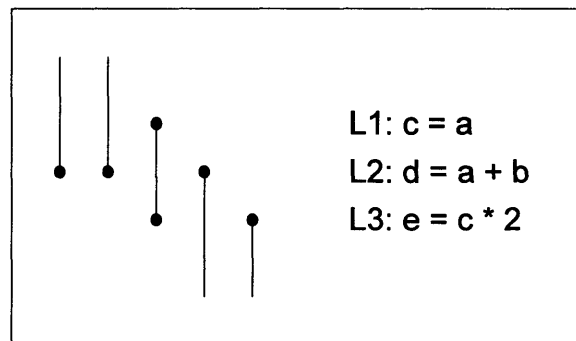
Creating larger supertraces is the topic of chapter six. We investigate two optimizations; basic block cloning and loop unrolling. Algorithms and examples are presented for each of the optimizations. We discuss the applicability of the optimizations when used with each of our local register allocation algorithms. Areas of future research and investigation are also presented.

CHAPTER 2 RELATED WORK

Introduction

In this chapter we will discuss previous work pertaining to register allocation and several different methods to optimize the assignment of variables to registers. Many of the techniques presented do not use separate local and global register allocation phases; instead the algorithms rely on heuristic graph coloring approaches for global register allocation. Although our register allocation algorithms do not perform general graph coloring, it is important to understand the principles and designs of these methods in order to compare our techniques.

Register lifetime is the number of instructions where the value of a variable stored in a register is active. Figure 2-1 shows a code fragment and several register lifetimes.



REGISTER LIFETIMES
FIGURE 2-1

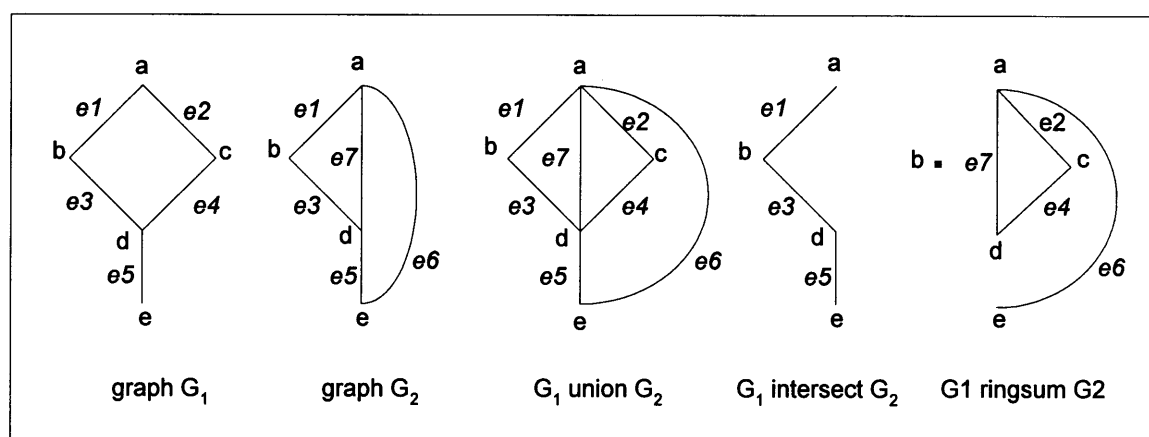
The lifetime of a variable begins with a definition and terminates after the last use of the variable. Variables a and b are *live* coming into statement L1, meaning the definition for the variables is in a previous statement. Because there is no use of variables a and b after statement L2, both variables are *dead* following statement L2. Dots represent the definition of variables c , d , and e at statements L1, L2, and L3 in the register lifetime chart. We show that variables d and e are live after statement L3 by the extended lines. Drawing a horizontal line across the register lifetimes after statement L1 shows that a , b and c are live at the same time. We cannot use the same physical register for variables a , b , and c . The number registers required to store live variables at some point is often called the *register pressure*. In the following section we present an overview of graph theory. Techniques from graph theory enable us to study the register allocation process in greater detail.

Overview of Graph Theory

We begin this section with definitions pertaining to graph theory. A graph is a set of objects connected by edges; formally $G=(V,E)$ where the set of edges $E=\{E_1,E_2,E_3,\dots,E_n\}$ connect the objects or vertices $V=\{V_1,V_2,V_3,\dots,V_n\}$. Vertices V_i , V_j associated with edge E_k are *end vertices*. When a vertex, V_i is an end vertex of some edge, E_j then V_i and E_j are *incident* to each other. *Self-loops* represent the case where end vertices connected by an edge are the same vertex. Multiple edges connecting the same end vertices are *parallel edges*. Nonparallel edges are *adjacent* if they are incident to a common vertex. The number of edges incident to a vertex V_i is the *degree* of V_i or $degree(V_i)$. *Directed graphs* have the property that edges connecting vertices have only one direction. Given a pair of vertices (V_i,V_j) and a connecting edge E_k , it is only possible to go from V_i to V_j or from V_j to V_i depending upon the direction associated with the edge. *Undirected graphs* do not have a direction associated with the edges. *Connected graphs* have paths between any two vertices.

Dividing graphs into smaller graphs creates *subgraphs*. A subgraph g must contain only vertices and edges present in graph G . *Edge disjoint subgraphs* occur when two or more subgraphs do not share a common edge. Subgraphs of a graph that

do not share vertices are *vertex disjoint subgraphs*. We can apply set operations of union, intersection and ring sum to graphs. These operations make it possible to join and decompose graphs. The union of two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ generates a third graph $G_3=(V_3,E_3)$ such that $V_3=V_1\cup V_2$ and $E_3=E_1\cup E_2$. The *intersection* of two graphs, written as $G_1\cap G_2$ create a new graph containing only the vertices and edges that are in both G_1 and G_2 . The ringsum of two graphs, written as $G_1\oplus G_2$, generates a graph consisting of vertices $V_1\cup V_2$ and edges that are in either G_1 or G_2 , but not both. *Deletion* of a vertex from graph G , denoted $G-V_i$, removes vertex V_i from graph G . Deletion of a vertex also implies the deletion of all edges incident on the vertex[8]. If E_j is an edge in graph G , then $G-E_j$ is a subgraph of G obtained by removing E_j from G . Deletion of an edge does not imply deletion of its end vertices. We can say $G-E_j=G\oplus E_j$. Figure 2-2 shows examples of union, intersection, and ringsum for graphs G_1 and G_2 .



GRAPH OPERATIONS
FIGURE 2-2

Graph traversal follows incident edges between end vertices. The *path* of a traversal is the sequence of vertices and edges used to go from the starting vertex to the ending vertex. A *simple* or *elementary* path contains only unique vertices. If the same vertex appears more than one time in the traversal, we use the term *walk* to describe the sequence of vertices and edges. Using graph G_1 from Figure 2-2 the sequence $\{a e1 b e3 d e5 e\}$ is a path from vertex a to vertex e . The sequence $\{a e1 b e3 d e5 e e6 a\}$ from graph G_2 in Figure 2-2 is a walk because vertex a appears more than one time. The number of edges traversed in a path is the *length* of the path. A

circuit is a closed nonintersecting walk in which no vertex except initial and final vertex appear more than once.

Graphs with a finite number of vertices and edges are *finite graphs*. If there is at least one path between every pair of vertices in graph G , the graph is *connected*. If a connected graph has n vertices and $n-1$ edges, the graph is *minimally connected*. A *tree* is a connected graph without self-loops or parallel edges. The following equivalent definitions identify a tree, given a graph G with n vertices[8]:

- (1) G is connected and circuitless.
- (2) G is connected and has $n-1$ edges.
- (3) G is circuitless and has $n-1$ edges.
- (4) There is exactly one path between every pair of vertices in graph G .
- (5) Graph G is *minimally connected*.

Rooted trees have one vertex distinguishable from all other vertices in the graph. *Binary trees* have one vertex of degree two and all other vertices as having degree one or three.

Graph coloring is a technique to uniquely identify vertices in a graph such that no two adjacent vertices have the same color. There are many alternative colorings for most graphs. Optimal solutions to the graph coloring problem use the minimum number of colors for a graph $G=(V,E)$. It is possible to compute the minimum number of colors or *chromatic number*, $\chi(G)$ for a graph[16]. If d_{max} is the maximum degree of the vertices in a graph G , then $\chi(G) \leq 1 + d_{max}$.

Register assignment may be viewed a graph coloring problem. Using this paradigm, vertices in a graph represent variable lifetimes and edges connect vertices that are live at the same time. Graph coloring algorithms represent the number of registers with k and attempt to create a k *coloring* of the input. Performing global register allocation with such an algorithm has the disadvantage of being an NP-complete problem for general graphs[10]. Coloring nodes of a graph with a finite number of colors may require time exponential to the size of the graph.

Graph Coloring Algorithm

In the common approach, three phases are used in the graph coloring register allocation algorithm[5,6]. The first phase replaces variables with pseudo registers and constructs a register interference graph that identifies which pseudo registers are live at the same time. The second phase simplifies the graph if possible by removing vertices from the graph. The third phase accepts input from the second phase assigning physical registers to the pseudo registers. Each phase of the graph coloring algorithm is defined in greater detail below.

The input to the first phase is an intermediate language(IL) representation of a procedure or function. Variables used in the procedure are replaced with pseudo registers. Temporary symbolic registers are used when an operation requires several machine instructions. From this input, the first phase constructs the register interference graph.

The interference graph may be viewed as a matrix, where rows and columns represent symbolic (virtual) registers. If two symbolic registers are live at the same time, an entry is made in the matrix. The degree of each vertex in the interference graph is the sum of the number of entries across its row (or down its column). Construction of the interference graph can include architecture dependent information concerning register pairing, and general purpose versus special use registers[5,6]. The interference graph is an undirected graph.

It is possible to simplify the interference graph by coalescing nodes. During the assignment of pseudo registers, many register to register copies are performed. An example of unnecessary register to register copying is the assignment of a new pseudo register for arguments to functions. Function arguments are often copied to other pseudo registers immediately after function calls. The *subsumption* of pseudo registers assigns one register to two noninterfering vertices resulting from a register copy operation. It is necessary to update the IL and rebuild the interference graph after subsumption; this is computationally expensive. There are cases where subsumption is

possible but not permitted due to interference. Subsumption of two variables may create interference with a third variable because the third variable is defined within the live range of one (or both) of the subsumption candidates.

During the second phase, the register interference graph is simplified. Vertices with $degree(v) < k$ are removed from the graph and either colored immediately or placed on a stack for later coloring. The heuristic algorithm used to determine which vertex to remove from the graph may select the highest or lowest degree vertex. If a vertex is colored immediately, we say the algorithm uses a *largest first* or *smallest first* heuristic. If the algorithm places the removed vertex on a stack for later coloring, we say the algorithm uses either a *largest last* or *smallest last* heuristic. Choosing graph vertices with the largest degree last has been shown empirically to work best[2]. The degree for the neighbors of the removed vertex are reduced by one, creating more opportunity for graph coloring without the added cost of inserting *spill code*. Removing all vertices from the graph with fewer than k neighbors will result in an empty graph or a graph containing only vertices with more than k neighbors.

If the $degree(v) \geq k$, for all remaining vertices n in the interference graph, a vertex is selected for *spilling* by some cost heuristic. Spilling is the temporary storage of a variable outside the register set after each definition and the return of the variable to the register set before each use. Instructions are inserted into the intermediate program representation to save values to memory after each definition and fetch the value from memory before each use. Several heuristics may be used to identify which vertex to spill based on variable usage, loop nesting level, additional execution time incurred by the spill code, or the degree of the vertex. Vertices identified for spilling are removed from the interference graph. The second phase is complete when the register interference graph is empty.

If a vertex is selected for spilling, the IL form must be updated to reflect the changes. The process of updating the IL, building the register interference graph, and simplifying it is repeated until no spill code is required. This iterative process is computationally expensive. Given a sufficiently large pool of registers, most programs do not require spill code. Programs requiring spill code usually converge quickly[6].

Phase three accepts input from phase two, rebuilding the register interference graph and coloring each vertex. Using the *largest last* or *smallest last* heuristic, the register interference graph is reconstructed by popping a vertex from the stack and giving it a different color than its neighbors. With the *largest first* or *smallest first* heuristic, each vertex is colored in phase two as it is removed from the graph.

Techniques to optimize register allocation by graph coloring focus on delaying the insertion of spill code[3] or the selection of cost heuristics for spill decisions[4]. One optimization limits the number of variables spilled from each basic block[2].

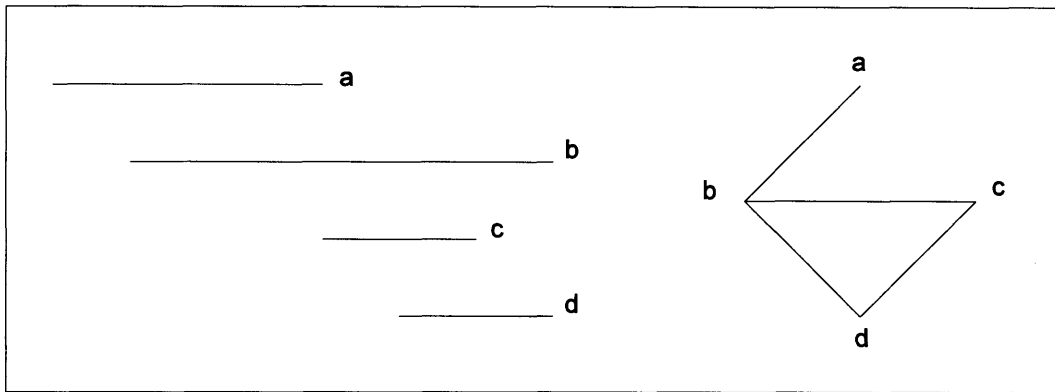
Register allocation by graph coloring fails to represent control flow information. The allocation algorithms must generate an interference graph, reduce the interference graph and generate global register assignments based on the reconstructed interference graph. If there are insufficient registers for simultaneously live variables, the intermediate form of the program must be modified and the entire process repeated. The local register allocation algorithms presented in chapters four and five have several advantages. We include control flow information in our representation making it possible to determine the live ranges with finer granularity. We do not need to rebuild the intermediate form of the program to spill registers.

Overview of Interval & Circular-Arc Graphs

Interval graphs show the intersection of a family of intervals along a real line. Vertices in the graph represent line segments, in our case they represent register lifetimes. Edges connect two vertices if there is any overlap between the segments. A sequence of vertices $[V_0, V_1, V_2, \dots, V_l, V_0]$ is a *cycle* of length $l+1$ if $V_{i-1}V_i \in E$ for $i = 1, 2, \dots, l$ and $V_lV_0 \in E$. A cycle is a *simple cycle* if $V_i \neq V_j$ for $i \neq j$. A simple cycle $[V_0, V_1, V_2, \dots, V_l, V_0]$ is *chordless* if $V_iV_j \notin E$ for i and j differing by more than $1 \pmod{l+1}$. *Comparability graphs* are undirected graphs where each edge can be assigned a one way direction resulting in an oriented graph (V, F) satisfying the following condition[9]:

$$ab \in F \text{ and } bc \in F \text{ imply } ac \in F \text{ where } (\forall a, b, c \in V)$$

We present an example of line segments and the derived interval graph in Figure 2-3.



Line Segments & Interval Graph
Figure 2-3

Properties from graph theory identify whether an undirected graph is an interval graph[9]:

- (1) The graph G , contains no chordless 4-cycles and its complement G' is a comparability graph.
- (2) The maximum cliques of G can be linearly ordered such that, for every vertex of G , the maximal cliques containing x occur consecutively.
- (3) An interval graph cannot branch in more than two directions, nor can it circle back onto itself.

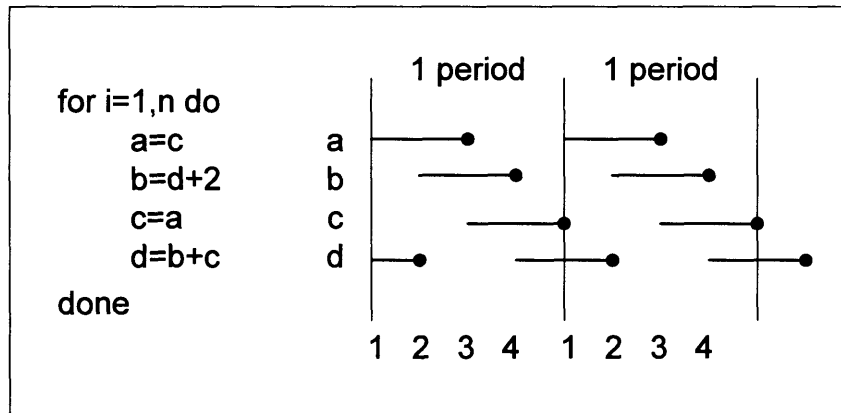
Circular-arc graphs present intervals around a circle. Every interval graph is a circular-arc graph, however not every circular-arc graph is an interval graph. The graph on the right side of Figure 2-5 shows a circular-arc graph. An undirected graph, $G=(V,E)$ is a circular-arc graph if and only if its vertices can be (circularly) indexed $V_1, V_2, V_3, \dots, V_n$ so that that for all i and j :

$$V_i V_j \in E \Rightarrow \{ \text{either } V_{i+1}, \dots, V_j \in \text{Adj}(V_i) \text{ or } V_{j+1}, \dots, V_i \in \text{Adj}(V_j) \}$$

if $i < j$, then V_{j+1}, \dots, V_i means $V_{j+1}, \dots, V_n, V_1, \dots, V_i$

Line segments used in a period or cyclic period graph can be assigned a beginning time index, t_i and an ending time index, t_j , where $t_i < t_j$. We use the notation $[t_i, t_j)$ to include the first endpoint and exclude the second endpoint or $[t_i, t_j]$ to include both endpoints. If all line segments repeat in a pattern, we can divide the family of segments

into *periods*. We represent a line segment in multiple periods as $P_1[t_i, t_j)$, $P_2[t_i, t_j)$, ..., $P_n[t_i, t_j)$. The graph in Figure 2-4 shows a period graph for the code fragment.



PERIOD GRAPHS
FIGURE 2-4

Cyclic period graphs permit line segments to loop back on themselves. We redefine the term *cyclic interval*[13] to *cyclic period*. Instead of representing segments in increasing periods, cyclic period graphs use arrows to denote line segments that wrap around endpoints. Cyclic period graphs use time index notation to represent line segments. The graph on the left side of Figure 2-5 is a cyclic period graph. The following definitions apply to intervals in cyclic period graphs[13].

- (1) A time t is covered by a line segment $S1:[t_i, t_j)$, if $(t_i \leq t < t_j)$, or by a line segment $S1':[t_i, t_j]$ if $(t_i \leq t \leq t_j)$, or by a cyclic period $S2:([t_i, t_j), [t_k, t_l])$ if $[t_i, t_j)$ or $[t_k, t_l]$ covers t .
- (2) Two segments $S1, S2$ overlap if there exists a time t that is covered by both $S1$ and $S2$.

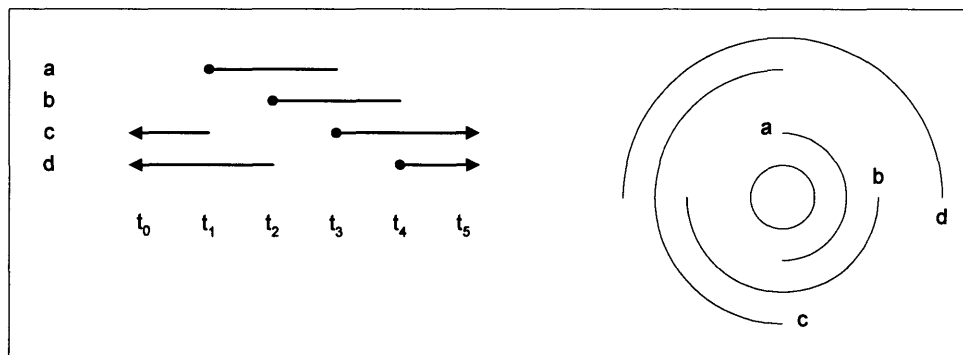
We redefine *hierarchical cyclic interval graph*[13] to *hierarchical cyclic period graph* in this work. Hierarchical period graphs permit nesting of cyclic period graphs. This family of graphs can represent conditional entry and exit of cyclic period graphs. Modeling complex control flow and nested iteration is the strength of hierarchical period graphs. The graph reads from left to right; increasing time indices place the segments in order. The graph identifies control flow and loops by drawing boxes around the line segments. Arrows on a line segment at the entry and exit of loop boundaries identifies a cyclic period. The appearance of nested boxes in a hierarchical cyclic graph illustrates

nested cyclic periods. The graph in Figure 2-6 shows an example of a hierarchical cyclic period graph.

Cyclic Period Graph Coloring Algorithm

The cyclic period graph algorithm uses control flow information about the loop nesting level of variables. Standard graph coloring algorithms do not consider the level of loop nesting where interference occurs. Standard graph coloring algorithms do not consider the length of interference (the number of instructions where variables are simultaneously live). This heuristic algorithm uses cyclic period graphs as an approach to register allocation. Cyclic period graphs are capable of recording the program structure and relative time where live ranges coincide, as shown in Figure 2-4. The first assignment statement in the loop, $a=c$, shows that c is live coming into the loop and that the live range for a does not carry around the loop. Subsequent iterations will use the loop carried value of c from statement three. The second assignment statement, $b=d+2$, defines b within the loop. The use of variable b in the fourth assignment statement, $d=b+c$, establishes a live range for b that is within the same iteration. The third and fourth assignment statements demonstrate variables that are live around the loop, c and d . Sequence numbers along the base of the graph represent instruction numbers.

In Figure 2-5 we show the cyclic period graph and circular arc graph representations for the program fragment from Figure 2-4.



CYCLIC PERIOD GRAPH & CIRCULAR ARC GRAPH
FIGURE 2-5

Variables carried around a loop create *cyclic line segments*. Cyclic period graphs represent live ranges across iterations of a loop. The arrows on line segments show loop carried variables. The baseline values t_0 - t_5 represent time points. Note that t_0 and t_5 are entry and exit points respectively; they do not represent executable instructions. Line segments represent the live ranges for variables.

Coloring cyclic period graphs begins by examining the number of live ranges present at a point in graph. An area in the graph where register pressure is high is a *fat spot*. The number of simultaneously live variables determines the *width* of the period graph. The width has a minimum, $W_{min}(G)$, and a maximum $W_{max}(G)$. The *fat cover* algorithm uses two phases to color the cyclic period graph. In the first phase the fat cover algorithm locates a set of nonoverlapping line segments that cover all the fat spots. Giving this set of nonoverlapping line segments the same color reduces the width of each included fat spot by one. The algorithm continues to perform this left-to-right search and coloring until the graph is empty or the only remaining line segments overlap. If m cyclic line segments exist, the first phase attempts to color each of the m cyclic segments with a unique color since the graph is colorable in $W_{max}(G)$ colors. If a fat cover exists, a right-to-left traversal colors the other members of the line segment set. Removing the colored line segment set F from the graph allows coloring the remaining graph in $k-1$ colors. Three theorems are the basis for this conclusion[13]:

- (2.1) Let G be a cyclic period graph containing no cyclic line segments. Then G is optimally colorable with $W_{max}(G)$ colors.
- (2.2) Let G be a cyclic period graph containing cyclic line segments. Then G is optimally colorable with $W_{max}(G) \leq k \leq W_{max}(G) + W_{min}(G)$ colors.
- (3.1) If a cyclic period graph G is colorable in $k + W_{max}(G)$ colors, then for each cyclic line segment I_c of G , there exists a fat cover for G relative to I_c , call it F , such that $G - F$ is $k - 1$ colorable.

The second phase uses a left-to-right algorithm to color the remaining periods. If the first phase succeeds in coloring the m cyclic line segments, then graph G is optimally colorable in $k = W_{max}(G)$ colors.

We redefine *Chameleon intervals*[13] as *chameleon segments* in this work. Chameleon segments provide temporary storage for line segments that are subject to

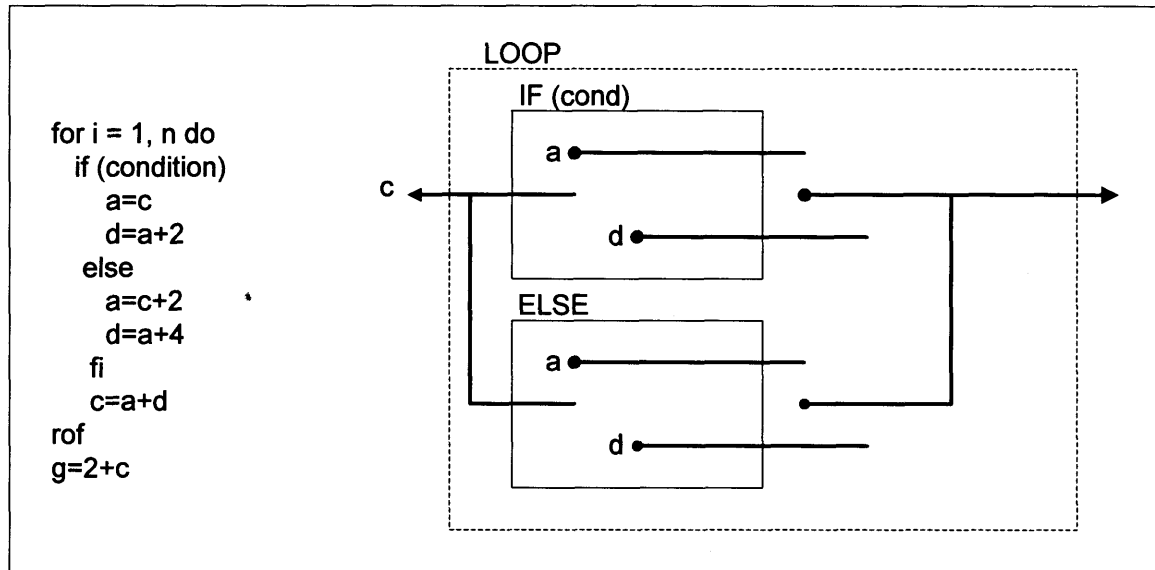
spilling otherwise. The name chameleon segment refers to cyclic line segments given a different color in the region of interference. Floating register values is less expensive than spill code because register to register moves do not require external memory accesses. The dependence of values may require the use of a temporary register to store the floated value. This algorithm introduces chameleon segments in situations where cyclic segments do not have a fat cover.

An alternative to the fat cover algorithm called the *sweep and split* algorithm. The algorithm uses a left to right pass to process a cyclic period graph. At each time index i , the graph has a measurable width of $W_{max}(G,i) \leq k$. During time $i+1$, two possibilities exist; the graph continues to have $W_{max}(G,i) \leq k$ or $W_{max}(G,i) = k'$, where $k' > k$. In the first case the algorithm increments the time index. In the latter case the algorithm must select $k' - k$ line segments to split by introducing spill code. The algorithm splits line segments with the furthest next use from i at the point of overlap. Time index values used to compute the furthest next use are readily available from the cyclic period graph. Secondary heuristics determine which line segment to split when there is a choice between multiple candidates. The secondary heuristics select which line segment to split by comparing the number of load and store instructions in each of the candidate segments. Secondary heuristics choose segments containing only load instructions before segments with load and store instructions. If all line segments require a store instruction, then the selection heuristics consider whether line segments extend beyond loop boundaries.

Hierarchical Period Graph Algorithm

We redefine the term hierarchical interval graph[13] to hierarchical period graph in this work. This algorithm extends the cyclic period graph algorithm to cover nested loops and conditional statements. Processing nested loops begins with the innermost loop, then the next outer loop, and so forth. Conditional statements have separate, parallel period graphs within the cyclic period graph, as shown in Figure 2-6. As with the other period graph algorithms, line segments represent the live ranges of variables. The graph in Figure 2-6 demonstrates the joining of line segments that are live beyond the end of a conditional statement. The algorithm also joins line segments for variables

carried around the loop. Line segments connected on inputs represent the same value; this is of critical importance for loop carried variables.



HIERARCHICAL PERIOD GRAPH
FIGURE 2-6

The algorithm uses techniques from the cyclic period graph algorithm in two phases. During the first phase (bottom-up), two steps are repeated[13].

- (1) Solve the innermost nested construct (either loop or conditional). In the case of a loop it is already a cyclic period graph. In the case of a conditional, create the proper cyclic period graph by joining the input and output variables that are common to both the IF-THEN and ELSE parts of the conditional.
- (2) Given the solution from 1, replace the nested structure with simple intervals.

The second phase is a top-down register allocation using chameleon segments to resolve cyclic periods entering and exiting conditional. An alternative to introducing chameleon segments is to repeat the first phase using additional coloring constraints.

Live Range Splitting

Live range splitting is possible during local or global register allocation to divide a register lifetime into two or more lifetimes. The graph coloring register allocation algorithm treats the new live ranges as variables. The interference graph requires

updating to reflect the new live ranges. Creating short live ranges provides greater flexibility in assigning virtual registers to physical registers. Increasing the number of live ranges has the potential of reducing the amount of spill code.

Live Range Splitting Algorithm

This algorithm defines a *live range* as a set of contiguous basic blocks in a control flow graph where a variable or virtual register is live. Definitions of the variable inside the live range do not reach outside the live range. Definitions of the variable that occur outside the live range do not reach a reference of the variable inside the live range. The algorithm calculates the execution time savings for each live range in a procedure or function. The algorithm then assigns a unique color to each live range using the greatest estimated time savings to order the live ranges. Color assignment halts when there are no unique colors for unassigned live ranges. For each uncolored neighboring live range conflicting for the register pool, the live range, lr splits in the following manner.

- (1) A new vertex is created in the interference graph for the new live range, lr_1 . The first basic block added to lr_1 is a definition block from lr , preferably an entry point to lr . This basic block is removed from lr .
- (2) Adjacent basic blocks are added to lr_1 from lr until the number of colored neighbors is one less than the total number of available colors. Updating neighbors in the interference graph is required when moving a basic block from lr to lr_1 . Adding basic blocks to lr_1 in this fashion creates the largest possible live range that is colorable.
- (3) If lr_1 is complete and has fewer than the maximum allowable neighbors, it is placed in a pool of low degree vertices. After removing basic blocks from lr , it is possible that it has fewer than the maximum neighbors, in which case it is also added to the low degree vertices. Basic blocks in the low degree vertex pool are not considered for live range splitting.
- (4) If lr or lr_1 have more than or equal to the maximum number of neighbors that are colored after splitting, the estimated cost savings is computed for lr and lr_1 . The live range with the greatest cost savings is assigned a color. Splitting is repeated on lr until it has one less than the maximum number of colored neighbors. Computing the estimated cost in this step is based on "...the total number of occurrences of the variable in the live range, weighted by loop nesting depths and normalized by the length of the live range." [7]

- (5) Vertices that were previously in the low degree pool may need to transition into the high degree pool because of live range splitting if it is now a neighbor to both lr and lr_1 .

Finding a vertex in the high degree subgraph that has the greatest execution time savings determines which live range splits. The cost of building the interference graph makes this algorithm computationally expensive. The complexity for this algorithm is $O(k(l - k))$; where k represents the maximum number of registers (colors) and l represents the number of live ranges[7].

We use a form of live range splitting in our modified supertrace register allocation algorithm. We chose to implement live range splitting by renaming variables. Chapter 5 contains the details of our implementation.

Multiple Register Operands

Language constraints (C unions, volatile variables and pointer addressing) and architectural support complicates the decision of which values to store in registers. Register allocation uses virtual registers for every static variable except unions, global data, pointer targets, array elements or other data requiring address computation. This algorithm addresses the problem of how to allocate registers or group of registers efficiently when instruction operands do not fit in a single register. We do not address the problem of multiple register operands in our experiments, though consideration for this topic is necessary in a compiler. Target architectures have limitations imposed by their design that system software (assemblers, compilers, linkers, ...) must deal with to create optimal register allocations.

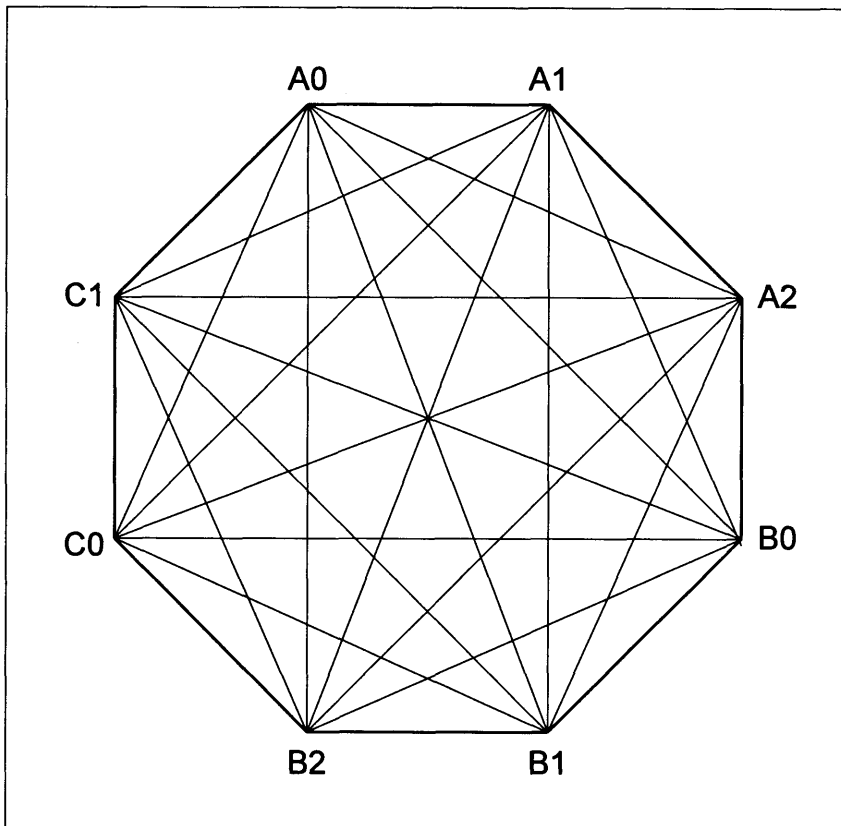
Dealing with processors that can address multiple registers for an operand does not fit into the typical graph coloring algorithm. Allocation for single and multiple register operands must occur at the same time[15]. Pre-allocating instructions that use multiple register operands results in artificially enhancing the priority of the instruction. Allocating multiple register operands after single register operands extends the schedule because of the difficulty in finding the required word and double word alignment (register number divisible by 2 or 4 respectively).

Groups of registers used for multiple register operands are a pair, triple or quad. The register allocator can not assign members of the register cluster to the same physical register. Cluster alignment is another restriction. In an eight register (R0..R7) processor, pairs could be assigned to R0-R1, R2-R3, R4-R5, and R6-R7. Assignment of triples and quads can occur at R0 or R4. Assigning a pair and two triples at the same time is not possible due to alignment constraints. Members of a register cluster will interfere with members of another cluster if both virtual registers are assigned to the same physical register. Pruning the graph and simplifying interference edges with normalization solves the problem of how to assign the register clusters onto the physical registers. Determining the order of cluster removal is based on *weighted_degree* for each cluster computed as:

$$\text{weighted_degree} = \text{constraints} * \text{weight_factor}_{\text{cluster_size}(c)} + \text{weight_bias}_{\text{cluster_size}(c)}$$

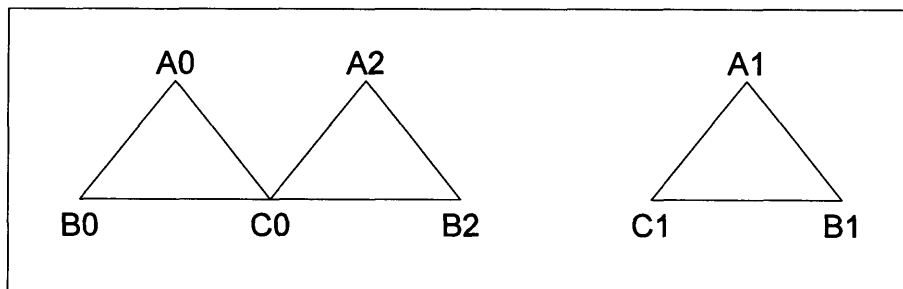
Constraints placed on a cluster are not equal to the constraints placed on graph vertices using the original formula for computing the degree of a vertex. The constraint value for a cluster is the number of interference edges to other clusters. Each cluster has a *weight_factor* based on the number of words required to store the cluster. Depending upon instruction argument size, Nickerson uses the values 1, 2, 4, and 4 as *weight_factors* for clusters of word size 1, 2, 3, and 4 respectively. Adding a *weight_bias* insures a positive value for the *weighted_degree* of clusters with few interference edges. The first clusters removed from the graph are unconstrained clusters with the largest *weighted_degree*. Heuristics based on the minimum cost, area, and *weighted_degree* determine the order for removal of clusters.

Instead of inserting spill code for constrained clusters with $\text{degree}(c) > k$, the cluster is *pruned* from the graph and the spill decision is delayed until the coloring phase[3]. It is possible that the coloring phase will be able to assign an unused color to the cluster that was not apparent during the removal of clusters. Normalization of the interference graph reduces the number of edges depicting implicit constraints. Consider two triples (A0,A1,A2), (B0,B1,B2), and a pair (C0,C1) as shown in Figure 2-7A with all implicit edges.



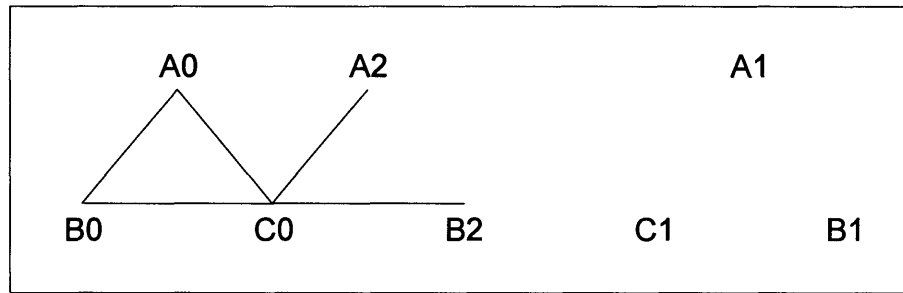
GRAPH OF TWO REGISTER TRIPLES AND A REGISTER PAIR
FIGURE 2-7A

Removing implicit edges as shown in Figure 2-7B simplifies the graph. The graph edges representing implicit conflicts between members of the same cluster, or between the members of clusters that cannot occupy the same register due to alignment are removed in this step.



GRAPH WITHOUT IMPLICIT EDGES
FIGURE 2-7B

Normalizing the graph removes synonymous interference edges. Note the lack of graph edges between $A1$, $B1$, and $C1$ in Figure 2-7c. The constraints between cluster members still exist through the relationship to the cluster boss. The interference graph contains the minimum information necessary to depict constraints between clusters after normalization.



NORMALIZED GRAPH WITHOUT IMPLICIT EDGES
FIGURE 2-7c

Delaying spill decisions until the coloring phase optimizes graph coloring. The heuristics select the least costly coloring when multiple colorings exist. When necessary, the coloring phase uses information from the three spill heuristics to decide which variable to spill. If the coloring step spills a variable, the intermediate form of the program requires modification to reflect the changes. The graph coloring algorithm iterates through the first and second passes to rebuild the interference graph and reduce it when modifications to the intermediate form take place.

Flow Graph Techniques

Algorithms discussed in this section offer different methods of allocating registers and dealing with spill code. Viewing the intermediate form of the program as a sequence of definitions and uses of variables controlled by selection and iteration is limiting. Increasing the granularity to *basic blocks* allows identification of *local* and *global* variables that we use in our research effort. The following algorithms abstract the tree into basic blocks or collections of straight line code separated by control operations. We present these techniques as an introduction to the following chapter.

Representing a program as a tree of *tiles* is one approach [4]. This algorithm addresses the problem of local register allocation by encapsulating basic blocks within loops and conditional control constructs into a *tile*. Using a control flow graph has the advantage of maintaining loop and conditional information available during register allocation. This information is useful in selecting infrequently executed areas in the program to insert spill code, should it be necessary. This technique uses a coarser granularity for local register allocation than our basic block algorithms. Heuristics used to identify tile boundary is very similar to the boundaries used in creating supertraces.

The authors' definition for a graph is $G = (B, E, start, stop)$ [4]. The set of basic blocks B , the set of edges between basic blocks is E and the unique single entry and exit points to the graph are *start* and *stop* respectively. Tile creation occurs in a bottom up traversal of the control flow graph. The algorithm assigns a tile to each loop and conditional structure. Each tile is disjoint from other tiles or is a proper subset of another tile. Initial coloring on each tile assigns pseudo registers in a local interference graph. Information from this coloring elevates to the *parent* tile. The interference graph for the parent tile incorporates each of the local interference graphs for its *children* into its interference graph. Each tile may create spill code at basic block boundaries using heuristics derived from local usage patterns. Following the bottom up traversal, a top down traversal assigns pseudo registers to the physical registers. Final coloring uses the delayed spill code decision technique[3].

A *Clique separator* is, "a completely connected subgraph whose removal disconnects the graph into two subgraphs." [11] This algorithm divides the program into code segments and colors each independently during a later phase. Final coloring of the individual cliques may involve register copying. The authors assert that using register copying eliminates the need for a local register allocation phase, due to the efficiency of the underlying algorithm. This algorithm addresses the problem of finding points in the flow graph of a program where creation of subgraphs can occur. This 'divide and conquer' technique is similar in principle to the creation of supertraces in our work. The clique separator algorithm differs from ours in that we do not use graph coloring, and we perform only local register allocation.

The structure of the program determines clique separators, not the structure of the interference graph[17]. The algorithm generates three sets during a top down traversal of the code segments. *CLIQUE* contains members of the current clique. *PRE* contains the live ranges that have already ended but overlap one of the members of *CLIQUE*. The set *POST* contains live ranges that have not begun, but overlap members of the set *CLIQUE*. The clique separator formed by members of *CLIQUE* is chosen if and only if it can be divided into disjoint sets *CLIQUE_{pre}* and *CLIQUE_{post}*. Valid separators have spans from *PRE* that do not overlap spans from *CLIQUE_{post}*, and the spans from *POST* must not overlap spans from *CLIQUE_{pre}*. The final requirement for selecting the clique separator is that the sets *PRE* and *POST* are not empty. The algorithm chooses clique separators so that the resulting subgraphs are smaller than the original graph creating less need for spill code.

CHAPTER 3

COMPILER FRAMEWORK

We describe flow graphs, graph trees and live variable analysis in this chapter. Data flow analysis is a technique to examine the flow of information through computer programs. Modeling programs using data flow analysis captures information from branches, joins, and loops. Use of data flow analysis permits identification and examination of variable live ranges. We present terms and concepts from data flow analysis pertinent to our research. We present live variable analysis techniques and the algorithms. Register allocation and register assignment terminology is presented as an introduction to subsequent chapters.

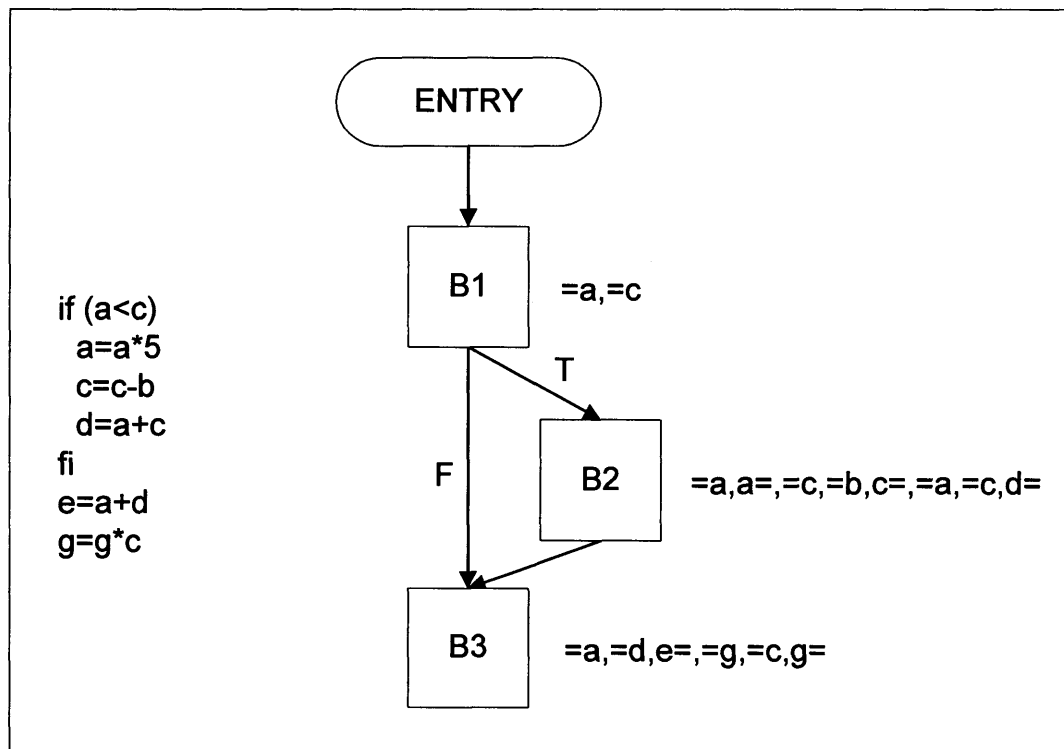
Data Flow Analysis

We can model a computer program using a *directed flow graph*, where nodes are computations and edges depict control flow by adding a unique entry point. Constructing the graph so that all nodes in the graph are reachable from a unique node, ENTRY, creates a *control flow graph* (CFG). The CFG of a subroutine captures branch and loop information necessary for our optimization techniques presented in later chapters. Representing selection, iteration, and sequence is possible by inserting the appropriate edges between nodes in the graph.

A variable reference in the flow graph of a computer program is either a *use* or a definition, *def*. If the variable reference establishes a new value for a variable, then the variable reference is a *def*. Variable references in the right hand side of an assignment statement or in conditional expressions are examples of uses.

Basic blocks contain consecutive statements. The only entry and exit points in a basic block are at the beginning and end of the basic block. Entry into a basic block under these conditions insures the uninterrupted execution of every statement in the basic block. Figure 3-1 shows an example of a conditional statement and the derived control flow graph. Associated with each basic block is a list of variable uses and defs derived from the statements.

Data flow analysis requires a single entry point to the CFG. The conditional statement at node *B1* compares the values of *a* and *c* to determine the whether to branch to node *B2* or continue to node *B3*. Graph edges marked T and F identifies the true and false conditional control paths. Basic block *B2* contains three assignment statements that execute when the condition tested in *B1* is true. Statements in *B3* will always execute. The directed flow graph also shows variable uses and defs for each basic block. The constant value 5 is not in the list of variables for *B2* because it is an immediate value.



ANNOTATED FLOW GRAPH
 FIGURE 3-1

Spanning Trees in Directed Graphs

A spanning tree is *rooted* at the *ENTRY* of a CFG. Spanning trees contain all vertices of the CFG and sufficient edges to permit a unique path from *ENTRY* to any vertex represented in the CFG. Four types of CFG edges exist in a spanning tree[16].

- (1) Edges contained in the spanning tree are called *tree edges*.
- (2) *Advancing (forward) edges* are edges $v \rightarrow w$ that are not tree edges but where w is a proper descendant of v in the spanning tree.
- (3) Edges $v \rightarrow w$ such that $v = w$ or where w is an ancestor of v are called *retreating (back) edges*.
- (4) *Cross edges* in the CFG are edges $v \rightarrow w$ such that w is neither an ancestor nor descendant of v in the spanning tree.

Given an edge in the CFG $v \rightarrow w$, we say that v is a predecessor of w and that w is a successor of v . We can create a list of successor vertices to V , $SUCC[V] = \{W \mid (V,W) \text{ is an edge}\}$; where $SUCC[V]$ is the list of vertices that are heads of edges with tail V . We create a *depth first spanning tree (DFST)* with a depth first search of the CFG. Figure 3-2 shows the *depth first search* algorithm.

```

Procedure Depth_First_Search(v)
  mark v "old";
  Numberpre(v) = n++
  for each vertex w on L[v] do
    if w is marked "new" then
      add (v,w) to T
      CALL Depth_First_Search(w)
    fi
  rof
  Numberpost(v) = p++
end

begin program
  T = ∅; n = p = 0
  for all v in V do
    mark v "new"
  rof
  Call Depth_First_Search(entry)
end

```

Depth First Search Algorithm
Figure 3-2

We use a left-to-right preorder numbering for basic blocks in the depth first spanning tree. Each basic block in the spanning tree has a unique left-to-right preorder number, $\text{Number}_{\text{pre}}(\cdot)$. Comparing the preorder numbers for two basic blocks v and w , $\text{Number}_{\text{pre}}(v) < \text{Number}_{\text{pre}}(w)$, will not discern whether v is an ancestor of w in the DFST or if v is 'to the left' of w . Determining the ancestor relationship between basic blocks requires adding an additional sequencing number to each basic block in the depth first spanning tree. Performing a right-to-left bottom-up traversal of the DFST generates $\text{Number}_{\text{post}}(\cdot)$ for each basic block. Our equation for the ancestor function is:

$$\text{ancestor}(v,w) = \text{Number}_{\text{pre}}(v) < \text{Number}_{\text{pre}}(w) \wedge \text{Number}_{\text{post}}(v) > \text{Number}_{\text{post}}(w)$$

In the context of a DFST we have additional properties associated with the vertices v and w . We can show v is an ancestor of w in the DFST iff $\text{ancestor}(v,w)$ is true.

Control Flow Constructs

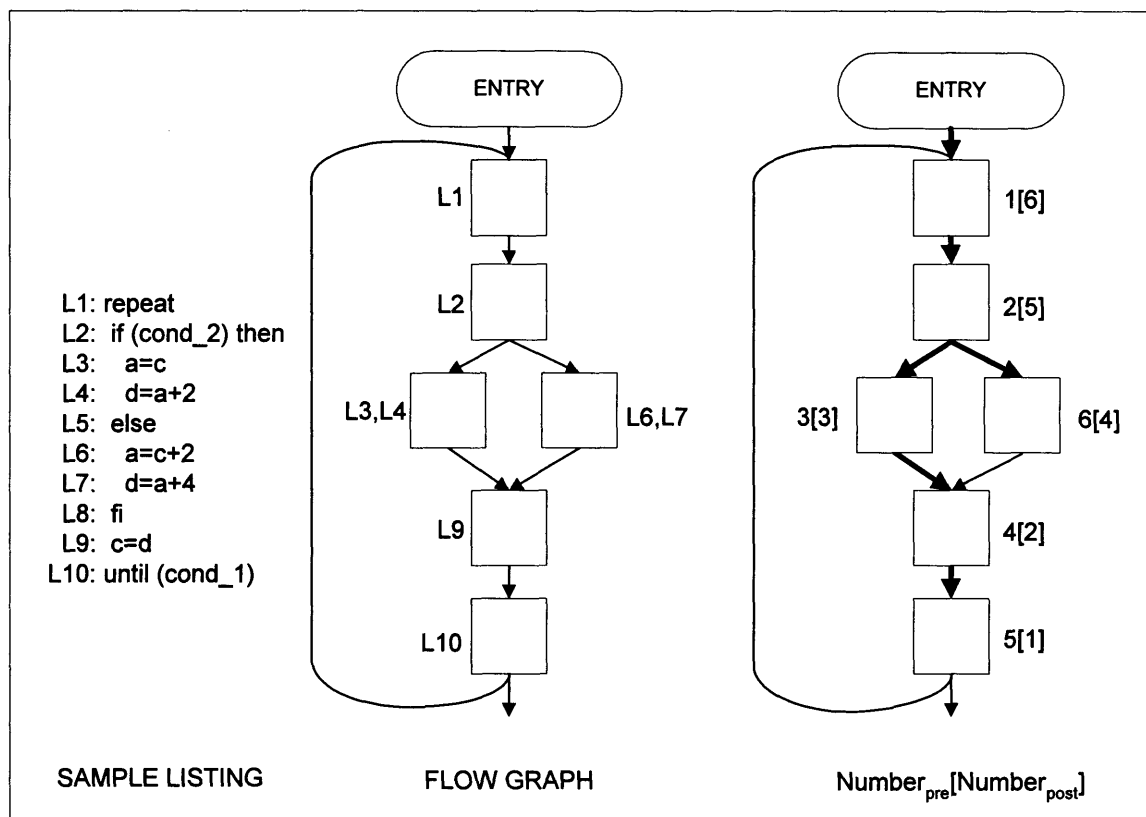
Cycles in control flow graphs permit iteration among one or more basic blocks. We define a *natural loop* to have two properties:

- (1) *There is one basic block that permits entry into the loop, known as a loop header.*
- (2) *There is an edge in the graph to the loop header that permits iteration.*

Detecting loops in reducible graphs requires finding the target node of a successor edge that points to a DFST ancestor. Nodes may loop back to themselves; such references are *self loops*. It is possible to construct loops with more than one member or have loops nested inside other loops. In the case of nested loops, the *inner* loop is the most deeply nested loop within the loop construct[1].

Figure 3-3 shows a program fragment and its CFG. The alphanumeric strings next to the basic blocks in the first CFG represent program statements. The basic blocks in the second CFG have a preorder and postorder traversal numbers. The DFST algorithm generates the preorder numbering, $\text{Number}_{\text{pre}}$, in the second CFG shown in Figure 3-3; tree edges are shown in bold. Statements L3, L4 and L6, L7 merge into the respective basic blocks because the assignment statements execute without interruption. Arrows denote control flow between the entry point and the basic blocks. The natural loop

formed between L10 and L1 represents the control structure of the “repeat - until” loop. Links between basic blocks accurately represent the input program. This example demonstrates iteration, selection and sequence.



SPANNING TREE NUMBERING
FIGURE 3-3

Live Variable Analysis

In this chapter the granularity used to determine whether a variable is local or global is the basic block. We use the live variable analysis algorithm in our search for local variables. The terms *local* and *global* in the context of live variable analysis have the following definitions. A variable is *global* if a definition exists in a basic block and has a reference in another basic block. When a variable is global, it is *live-out* from the basic block where it is defined and *live-in* to the basic block where the first reference of the variable is a use. The use of variable v is a *local use* when v is not a member of the LIVEBEFORE set of the basic block. If every use of variable v is local, then v is *basic block local*. Given a flow graph of several basic blocks, local register allocation is

possible for any variable that is not a member of the LIVEBEFORE set of any basic block. The following equations compute LIVEBEFORE and LIVEAFTER for variables in basic blocks.

$$\text{LIVEAFTER}(V) = \cup (\text{LIVEBEFORE}(S), \text{forall } S \text{ in } \text{SUCC}(V))$$

$$\text{LIVEBEFORE}(V) = \text{use}(V) \cup (\text{LIVEAFTER}(V) - \text{def}(V))$$

The live variable analysis algorithm, shown in Figure 3-4, operates on variable uses and definitions contained in an ordered list of instructions for each basic block. Each of the sets (LIVEBEFORE, LIVEAFTER, UPWARDUSES, NOTKILLED) is a separate bit vector in each basic block. In this algorithm it is necessary to traverse the list of instructions in the basic block in a last to first order. Traversing the instruction list in reverse order finds variable uses live coming into the basic block, also called *upwardly exposed uses*. The order for visiting basic blocks is not important.

The LIVEBEFORE vector contains variables used before they are defined or redefined. The UPWARDUSES vector contains variables from the current basic block that are first referenced as a use. Variable definitions cause removal of the variable from the UPWARDUSES bit vector. Earlier uses of variables update the UPWARDUSES vector. Initialization of the NOTKILLED bit vector assigns all variables present in the basic block. Definitions found in the instruction list remove the variable from the NOTKILLED bit vector. During the second pass of the algorithm, the LIVEAFTER bit vector becomes the union of the LIVEBEFORE bit vectors for all basic block successors. The algorithm shows how the NOTKILLED and UPWARDUSES vectors update the LIVEBEFORE bit vector.

The algorithm accepts a CFG as input. Each basic block contains bit vectors initialized and assigned based upon the presence or absence of uses and defs. We then update the LIVEAFTER and LIVEBEFORE vectors by iterating over the basic blocks in bottom up traversal. The second pass does not require a bottom up traversal, however it is more efficient and minimizes additional iterations over all the basic blocks. Several iterations through the CFG may occur in the second part of the live variable analysis algorithm before the LIVEAFTER bit vector stabilizes.

```

PROC LiveVar(GRAPH)
for all basic blocks, V in flowgraph, GRAPH do
  LIVEAFTER(V,*) = 0
  UPWARDUSES(V,*) = 0
  NOTKILLED(V,*) = 1
  for all use/def references, UD in V in reverse order do // upward exposed
    uses/kills
    NUM = SYMTABNUMBER(SYMBOL(UD))
    if UD is a use then
      UPWARDUSES[V,NUM] = 1 // set use bit
    elseif UD is a def then // not preserved, killed
      NOTKILLED[V,NUM] = 0
      UPWARDUSES[V,NUM] = 0
    fi
  rof
  LIVEBEFORE(V,*) = UPWARDUSES(V,*) // initialize livebefore vector
rof
repeat
  CHANGED = FALSE
  for basic blocks, V of flowgraph, GRAPH in reverse order do
    TVECT(*) = 0
    for all SUCC(V), SV do
      TVECT(*) = TVECT(*)  $\vee$  LIVEBEFORE(SV,*)
    rof
    if TVECT(*)  $\neq$  LIVEAFTER(V,*) // no convergence of LIVEAFTER vector yet
      LIVEAFTER(V,*) = TVECT(*)
      LIVEBEFORE(V,*) = (TVECT(*)  $\wedge$  NOTKILLED(V,*))  $\vee$  UPWARDUSES(V,*)
      CHANGED = TRUE
    fi
  rof
until not CHANGED
END LiveVar

```

LIVE VARIABLE ANALYSIS ALGORITHM
FIGURE 3-4

We present an example in Table 3-1 and Table 3-2 using the live variable analysis algorithm to show the LIVEBEFORE and LIVEAFTER sets for the program fragment from Figure 3-3. First pass operations, shown in Table 3-1, compute the UPWARDUSES, NOTKILLED and LIVEBEFORE bit vectors. Processing instructions in each basic block from last to first to preserves information about variables that are live coming into the basic block. Table 3-1 shows instructions arranged in the processing sequence used by the live variable analysis algorithm. Additional lines are present showing (re)initialization of the bit vectors during the first pass.

BLOCK	INSTRUCTION	UPWARDUSES	NOTKILLED	LIVEBEFORE	LIVEAFTER
1		{}	{a,c,d}	{}	{}
2		{}	{a,c,d}	{}	{}
3	=a,d= =c,a=	{a} {c}	{a,c} {c}	{c}	{}
4	=d,c=	{d}	{a,d}	{d}	{}
5		{}	{a,c,d}	{}	{}
6	=a,d= =c,a=	{a} {c}	{a,c} {c}	{c}	{}

The second pass, shown in Table 3-2, will iterate until the LIVEAFTER bit vector contains the same variables as its successor's LIVEBEFORE bit vector. It is important to note that computing the LIVEAFTER bit vector must precede computing the LIVEBEFORE bit vector. During the convergence step in live variable analysis the algorithm visits basic blocks in reverse, or $\text{Number}_{\text{post}}$ order. In this example the algorithm will iterate twice during convergence, no changes to the LIVEBEFORE or LIVEAFTER bit vector occur during the second iteration. Table 3-2 shows two series of computations, with no changes after the second series. The order of basic blocks in Table 3-2 reflects the sequence in which processing occurs. In this example the global variables are *c* and *d*. The variable *a* is local and basic block local by our previous definition.

BLOCK	LIVEAFTER _{t0}	LIVEBEFORE _{t0}	LIVEAFTER _{t1}	LIVEBEFORE _{t1}
5	{}	{}	{c}	{c}
4	{}	{d}	{c}	{d}
3	{d}	{c}	{d}	{c}
6	{d}	{c}	{d}	{c}
2	{c}	{c}	{c}	{c}
1	{c}	{c}	{c}	{c}

Register Assignment

This phase determines which physical register will store the variables or virtual registers identified during register allocation[1]. There are many ways to select the specific register given the constraints of how many general purpose registers are

available. Another consideration is whether to use register pairs, triples or quads to store multiple word values. Many of the register assignment techniques presented in Chapter 2 use a one pass algorithm to achieve global register assignment. In this section we describe the benefits of having a local and a global register assignment pass compared to a single global register allocation policy.

Assigning physical registers to the values in virtual registers requires information from live variable analysis. Our first goal is to identify live virtual registers within a basic block. If the live range of a virtual register is wholly within a single basic block, meaning the virtual register is not live coming into or leaving the basic block, we can use a linear time algorithm to assign a physical register to that value. Assignment of physical registers for virtual registers with live ranges across basic block boundaries is by a global assignment algorithm executing in polynomial time. The benefit of using a local register assignment is that the algorithm is fast, and it removes variables from global assignment which is slow. Global register assignment is left with fewer variables and will take less time. Subsumption and dead code elimination are optimizations that apply to local and global register assignment.

CHAPTER 4

LOCAL REGISTER ALLOCATION

We present detailed information about two basic block register allocation algorithms in this chapter. We describe the relative merits and shortcomings of each algorithm. We examine how the local register allocation algorithms use live variable analysis. These algorithms assign registers for variables that are local to a basic block.

Backward Pass Algorithm

In this register allocation algorithm, we associate two properties with each variable. We define the property STATE with values $\{local, global\}$ to represent live variable analysis results for variables in the symbol table. Each basic block contains its own local list of variable uses and defs and a property called LIVESTATE with values $\{live, dead\}$ to identify if a variable is active at some point in the basic block. Live variable analysis provides necessary information about local and global variables to the initialization and register assignment passes of the backward pass register allocation algorithm.

Processing the control flow graph involves one traversal of the basic blocks. The order of visiting basic blocks in the control flow graph is not important because we have already computed which variables are local and global during live variable analysis. For each basic block the backward pass register allocation initializes LIVESTATE to *dead* for all local variables.

Register allocation and assignment follow initialization. This algorithm processes instructions in each basic block from last to first. Our first test is for definitions that are

live due to a later use. Live variable definitions receive registers before operands. The algorithm removes local variable definitions without subsequent uses as dead code. Variable uses local to the basic block receive a register. Local uses have their LIVESTATE set to *live* after being assigned a register. Free registers are kept on a stack, the most recently freed register is the first one used. Using a stack for free registers is a requirement for optimality. In Figure 4-1 we show the backward pass register allocation algorithm.

```

Procedure BackwardPass(GRAPH)
  for all variables, V1 of symbol table do
    if STATE(V1) == local then
      LIVESTATE(V1) = dead
    fi
  rof
  for all basic_blocks, B of flowgraph, GRAPH do
    for all instructions, I of B in reverse order do
      for all variables, V1 of I do
        if DEF(V1) ^ STATE(V1) == local then
          if LIVESTATE(V1) == dead then
            CALL DeadCode(I,V1)
          else
            CALL FreeReg(V1)
            LIVESTATE(V1) = dead
          fi
        else if USE(V1) ^ STATE(V1) == local then
          CALL AllocReg(V1)
          LIVESTATE(V1) = live
        fi
      rof
    rof
  rof
END BackwardPass

```

BACKWARD PASS BASIC BLOCK REGISTER ALLOCATION ALGORITHM
FIGURE 4-1

We now present an example of register allocation using the backward pass algorithm. The code fragment represents instructions stored in a single basic block. Live variable analysis of the instructions identify variables (b, d, g) as global. Variables (a, c, e, f) are local and assigned to registers. Table 4-1 shows the instructions in reverse order, the order used by the backward pass algorithm. We show registers live coming into the instruction on the same line as the statement number. Register assignments to the operands are on the next line unless the operands are constants or global variables. The last line used for each instruction replaces the target with a

register except where the target is a global variable. We compare the input to the results in Table 4-2, with the instructions in sequential order.

Stmt. #	Instructions	Variables				Registers		
		a	c	e	f	R1	R2	R3
i7	g=d+f				R1	f		
	g=d+R1				R1	f		
i6	f=e+d*c				R1	f		
	R1=e+d*c							
i5	R1=R1+d*R2		R2	R1		e	c	
	d=a+2		R2	R1		e	c	
i4	d=R3+2	R3	R2	R1		e	c	a
	e=g/2	R3	R2	R1		e	c	a
i3	R1=g/2	R3	R2				c	a
	a=a+c	R3	R2				c	a
i2	R3=a+c		R2				c	
	R3=R3+R2	R3	R2				c	a
i1	c=a+3	R3	R2				c	a
	R2=a+3	R3						a
i1	R2=R3+3	R3						a
	a=b+d	R3						a
	R3=b+d							

Stmt. #	Original Program	Backward Pass Results
i1	a=b+d	R3=b+d
i2	c=a+3	R2=R3+3
i3	a=a+c	R3=R3+R2
i4	e=g/2	R1=g/2
i5	d=a+2	d=R3+2
i6	f=e+d*c	R1=R1+d*R2
i7	g=d+f	g=d+R1

Forward Pass Algorithm

This basic block register allocation algorithm traverses the basic blocks in the CFG in a top down fashion. Live variable analysis determines basic block local and basic block global variables. We use STATE, with the values $\{local, global\}$ to represent live variable analysis results for each variable. This algorithm allocates registers for the

targets (defs) after operands (uses) to maintain the data dependence relations[8] between statements in the basic block. Figure 4-2 shows the forward pass basic block register allocation algorithm.

```

Procedure ForwardPass(GRAPH)
for variables, V1 in symboltable do
  LIVESTATE(V1) = dead
rof
for all basic blocks B, of flowgraph, GRAPH do
  for all instructions, I of B in reverse order do // BEGIN PASS #1
    V2 = target(I)
    if (V2) ^ STATE(V2) == local then
      if LIVESTATE(V2) == dead then
        CALL DeadCode(I,V2)
      else
        LIVESTATE(V2) = dead
    fi
  fi
  for all operands, V1 of I do
    LASTUSE(I, V1) = FALSE
    if STATE(V1) == local ^ LIVESTATE(V1) == dead then
      LASTUSE(I,V1) = TRUE
      LIVESTATE(V1) = live
    fi
  rof
  rof // END PASS #1
  for all instructions, I of basic block B do // BEGIN PASS #2
    for all operands, V1 of instruction, I do
      if STATE(V1) == local then
        CALL AllocReg(V1)
        if LASTUSE(I,V1) == TRUE then
          CALL FreeReg(V1)
        fi
      fi
    rof
    V2 = target(I)
    if (V2) ^ STATE(V2) == local then
      CALL AllocReg(V2)
    fi
  rof // END PASS #2
rof // for all basic blocks
END FORWARDPASS

```

FORWARD PASS BASIC BLOCK REGISTER ALLOCATION ALGORITHM
FIGURE 4-2

Before processing the instructions in the basic blocks, the algorithm sets LIVESTATE to *dead* for each variable in the symbol table. One iteration through the basic blocks determines the live ranges and assigns registers. Because the algorithm uses the granularity of a basic block, we set the LASTUSE of all variable *uses* to *FALSE*. The algorithm computes LIVESTATE from the instances of *uses* and *defs*. The algorithm's first pass goes through the list of instructions in reverse order removing dead code and identifies the which instance of a variable use is the last use. This pass identifies the live ranges of local variables. Register assignment for local variables occurs during the second pass.

In this algorithm, LIVESTATE has the values *{live, dead}* identifying whether a variable is live at a point in the basic block. Variable uses encountered in an instruction set the LIVESTATE to *live*. Variable defs in the instruction reset the variable LIVESTATE to *dead*. The STATE property has values *{local, global}* showing which variables are available for allocation. The LASTUSE property has values *{TRUE, FALSE}* identifying the instance of a variable that terminates the variable live range. In this algorithm any variable with a STATE of *local* is local to the basic block and eligible for local register allocation. Table 4-3 shows the results of the first pass operations where variables (a, c, e, f) are local and variables (b, d, g) are global.

TABLE 4-3 FORWARD PASS BASIC BLOCK REGISTER ALLOCATION INITIALIZATION					
Stmt. #	Instructions	Variables LIVESTATE/LASTUSE			
		a	c	e	f
i7	g=d+f				L/T
i6	f=e+d*c		L/T	L/T	D/
i5	d=a+2	L/T			
i4	e=g/2			D/	
i3	a=a+c	D/ ^{DEF}			
i3	a=a+c	L/T ^{USE}	L/F		
i2	c=a+3	L/F	D/		
i1	a=b+d	D/			

Entries in Table 4-3 combine information from LIVESTATE and LASTUSE. The following symbols *{L,D}{T,F}* identify variable live ranges; where 'D' represents *dead*, 'L' represents *live*, 'T' represents *last_use* and 'F' is *not_last*. Statements in Table 4-3 reflect the processing order of the algorithm. Table 4-4 shows operations performed by

the second pass where register assignment takes place. Table 4-5 shows the input and output from the forward pass algorithm.

Stmt. #	Instructions	Variables				Registers		
		A	C	E	F	R1	R2	R3
i1	a=b+d	R1				a		
	R1=b+d	R1				a		
i2	c=a+3	R1	R2			a	c	
	R2=R1+3	R1	R2			a	c	
i3	a=a+c	R1	R2			a	c	
	a=R1+R2	∅	R2			∅	c	
	R1=R1+R2	R1	R2			a	c	
i4	e=g/2	R1	R2			a	c	
	R1=g/2	R1	R2	R3		a	c	e
i5	d=a+2	R1	R2	R3		a	c	e
	d=R3+2	∅	R2	R3		∅	c	e
i6	f=e+d*c		R2	R3			c	e
	f=R1+d*R2		∅	∅			∅	∅
	R3=R1+d*R2				R3			f
i7	g=d+f				R3			f
	g=d+R3				∅			∅

Stmt. #	Original Program	Forward Pass Results
i1	a=b+d	R1=b+d
i2	c=a+3	R2=R1+3
i3	a=a+c	R1=R1+R2
i4	e=g/2	R3=g/2
i5	d=a+2	d=R1+2
i6	f=e+d*c	R3=R1+d*R2
i7	g=d+f	g=d+R3

Conclusion

Allocating registers at the granularity of a basic block is a well defined and understood process. The two register allocation algorithms presented in this chapter achieve efficient solutions for local register assignment in basic blocks. The backward pass algorithm requires one iteration through the basic blocks and one iteration through the instructions. The backward pass algorithm does not require the LASTUSE property.

The backward pass register allocation algorithm is the algorithm of choice for basic block register allocation. The forward pass algorithm requires one pass through the basic blocks and two iterations through the instructions. We present the forward pass basic block register allocation algorithm because it is the basis for register allocation algorithms presented in the next chapter. Allocating registers for basic block local variables takes advantage of short register live ranges.

Basic block register allocation guarantees to use the minimum number of registers if there are sufficient registers for local variables, and if a stack is used for free registers. In the case where there are insufficient registers for local variables, spilling heuristics do not provide a solution with a minimal number of spills. Reordering instructions will change the demand for registers. Finding a solution for minimal register usage with instruction reordering is beyond the scope of our research.

CHAPTER 5

REGISTER ALLOCATION ACROSS BASIC BLOCKS

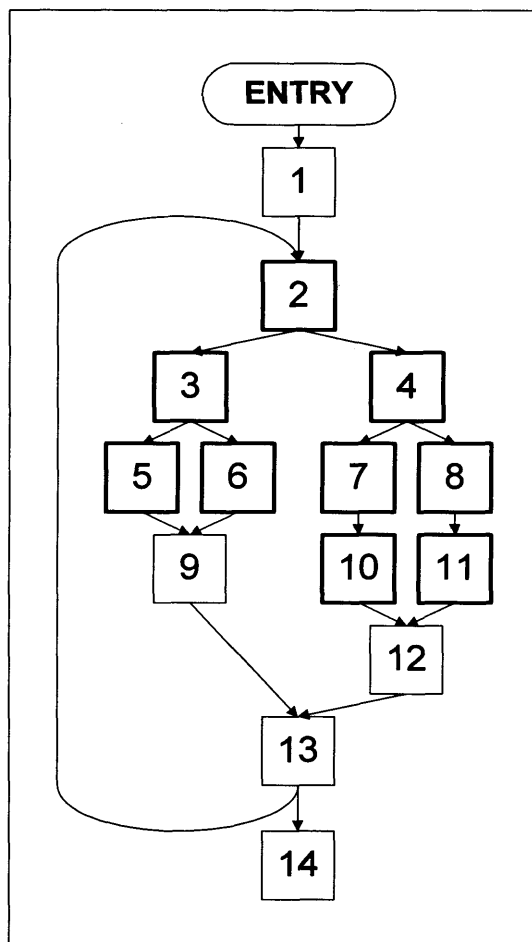
In this chapter, we present a technique for local register allocation that spans basic block boundaries. The algorithms locate and use trees of basic blocks, called *supertraces*, to extend live ranges of local variables. We discuss how the algorithms use control flow graph information to find the supertraces. We present the motivation and design criteria leading to the algorithms and their implementation.

Supertrace Formation

Control flow graphs contain trees of basic blocks. We define a supertrace as a region of the CFG with a single entry point where basic blocks form a tree. The root of a supertrace is a basic block located at entry of a CFG, the header of a natural loop, or a basic block located at a confluence point for two or more predecessors. Minimal trees consist of a single basic block matching the definition for a root of a supertrace. Our goal is to build supertraces with as many basic blocks as possible. We add basic blocks to a supertrace by following the successor links from each supertrace root. Basic blocks with a single predecessor link spanning off the supertrace head are members of the same supertrace. The path length from the supertrace root to confluence points in the CFG limits the size of a supertrace. Supertraces cannot grow beyond a confluence point (where basic blocks have two or more predecessors).

Figure 5-1 shows a CFG with the basic blocks of the large supertrace in bold. The large supertrace contains basic blocks 2, 3, 4, 5, 6, 7, 8, 10, and 11. Minimal supertraces exist for basic blocks 1, 9, 12, 13, and 14. Using this example, we can follow the creation of the large supertrace. Beginning at the entry to the CFG, the algorithm

marks basic block 1 as a supertrace root. Following the successor link from block 1 to block 2 we find a second successor link, the loop back edge. We cannot assign basic block 2 to the supertrace rooted at basic block 1 because basic block 2 has more than one predecessor. Because basic block 1 has no more successors, the supertrace is complete for this root. Basic block 2 meets the definition for a supertrace root, so we begin the generation of a second supertrace rooted at basic block 2. Following the successor links from 2 to 3, from 3 to 5, and 3 to 6 permit inclusion of these basic blocks into the supertrace because basic blocks 3, 5, and 6 have only one predecessor.



SUPERTRACE
FIGURE 5-1

Following the successor links from 5 to 9 and from 6 to 9 halts the creation of the supertrace on this branch because basic block 9 has two predecessors. The algorithm continues along the successor links from 4 to 7, from 4 to 8, from 7 to 10, and from 8 to

11, adding all basic blocks to the supertrace rooted at basic block 2. Our supertrace ends at basic block 12 because basic block 12 has more than one predecessor.

Supertrace Local Variables

In this section we introduce and define *supertrace local variables*. Within each supertrace, variables that are not live coming into or going out of a supertrace boundary are local to the supertrace. Variables that are not members of any supertrace head's LIVEBEFORE set are *supertrace_local*. The supertrace register allocation algorithm allows live ranges to cross basic block boundaries for supertrace local variables. Variables identified as LIVEBEFORE to a supertrace head are *supertrace global*. The global register allocation algorithm processes supertrace global variables. Following live variable analysis, we find supertrace local and supertrace global variables by making a pass through the basic blocks of each supertrace. Our algorithm looks for uses of variables within the supertrace that are not members of the LIVEBEFORE set of the supertrace head. The algorithm scans the supertrace looking at each variable in turn, assigning the state *supertrace_global* to variables that has any nonlocal use. The problem arises that a variable may be *supertrace_local* in one supertrace and *supertrace_global* in another. We resolve this conflict by making the variable supertrace global in all supertraces.

Forward Pass Supertrace Algorithm

The forward pass supertrace register allocation algorithm allocates registers for local variables. This algorithm is more complex than the basic block local register allocation algorithms in the previous chapter. There is an initialization step, a register allocation step, and a traversal step in this algorithm. The initialization step identifies the variable live ranges and deals with dead code elimination. The register allocation step assigns registers for local uses and defs of instructions. The traversal step in this algorithm is unique in that we save and restore the state of register assignment before visiting the next basic block in the supertrace. Successful register allocation based on supertraces requires assignment of all basic blocks in the CFG to supertraces. The allocation algorithm must visit all supertraces in the CFG. We present the main procedure for the algorithm in Figure 5-2.

```

PROC Forward_ST(CFG)
CALL Forward_Init(CFG)
for all basic blocks, B of flowgraph, CFG do
  if SupertraceHead(B) then
    CALL Supertrace_Pass(B)
  fi
rof
END Forward_ST

```

DRIVER ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-2

In this algorithm it is necessary to keep track of variables assigned to registers and registers assigned to variables. We use two data constructs REGREF and VARREF to keep this information. In VARREF we store information about registers assigned to variables. The REGREF data structure stores which variable is in a given register.

We show the initialization step for this algorithm in Figure 5-2A. Initialization of variable state information occurs before live range calculations or register allocation. The algorithm initializes REGREF and VARREF to null. Initialization visits all instructions within each basic block in a last to first order. This order permits finding the last use of a variable. LASTUSE and LIVESTATE store the computed live range of variables. The field LASTUSE can have the values {TRUE, FALSE}; the field LIVESTATE can have the values {*live*, *dead*}. We initialize LASTUSE of all operands to FALSE. Local variable definitions not used in subsequent instructions as an operand cause dead code removal of the instruction. Initialization sets LIVESTATE to *dead* for local definitions of a variable. Local variable uses not live beyond a basic block boundary (determined by LIVEAFTER membership) have LASTUSE set to TRUE and LIVESTATE set to *live*. These local uses mark the end of a variable's live range.

```

PROC Forward_Init(CFG)
  REGREF(*) = ∅
  VARREF(*) = ∅
  for all basic blocks, B of flowgraph, CFG do
    for all variables, V1 in symboltable do
      LIVESTATE(V1) = LIVEAFTER(B,V1) // {live, dead}
    rof
    for all instructions, I of B in reverse order do
      V1 = TARGET(I)
      if DEF(V1) ∧ STATE(V1) == local then
        if LIVESTATE(V1) == dead then
          CALL DeadCode(I,V1)
        else
          LIVESTATE(V1) = dead
        fi
      fi
      for all operands, V2 of I do
        LASTUSE(I,V2) = FALSE
        if STATE(V2) == local ∧ LIVESTATE(V2) == dead then
          LASTUSE(I,V2) = TRUE
          LIVESTATE(V2) = live
        fi
      rof
    rof
  rof
END Forward_Init

```

INITIALIZATION ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION
 FIGURE 5-2A

We show the register allocation step of the forward supertrace register allocation algorithm in Figure 5-2B. This modified forward basic allocator accepts a single basic block. We process variable uses before definitions to maintain data dependence relations between instructions in the basic block. Immediately after assigning a register for a variable use, we check if this is the last use of the variable and possibly free the register. The algorithm assigns variable definitions a register after processing instruction operands. Register allocation for each basic block uses the same sequence of operations shown in the forward pass basic block register allocator.

```

PROC Supertrace_Pass2(BASIC_BLOCK)
  for all instructions, I in BASIC_BLOCK do
    for all operands, V1 in I do
      if STATE(V1) == local then
        CALL AllocReg(V1)
        if LASTUSE(V1,I) == TRUE then
          VARREF[REGREF[V1]] = ∅
          REGREF[V1] = ∅
        fi
      fi
    rof
    V1 = TARGET(I)
    if DEF(V1) ^ STATE(V1) == local
      REG = GetReg()
      REGREF[V1] = REG
      VARREF[REG] = V1
    fi
  rof
END Supertrace_Pass2

```

ALLOCATION ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-2B

We show the traversal routine in Figure 5-2C. The traversal step is responsible for calling the register allocation routine for each basic block in the supertrace. After the register allocation step returns, the traversal routine saves the state of register assignments in SAVEREGS. We assign the same register state to all basic blocks that are successors and members of the supertrace. Transfer of the register state from SAVEREGS is only for variables live before the successor basic block.

If the successor basic block does not reference a variable that its predecessor defines, we free the register making it available. An example of where this occurs, taken from Figure 5-3, is the variable *d*. This variable is live coming out of basic block one because of the variable use in basic block five. Basic blocks two, three, and four have no uses of *d*. It is unnecessary to save the register assignment for variable *d* established in basic block one and have it present during register allocation of basic blocks two, three, and four. The register state saves variable *d* for basic block five because of the use in the first assignment statement.

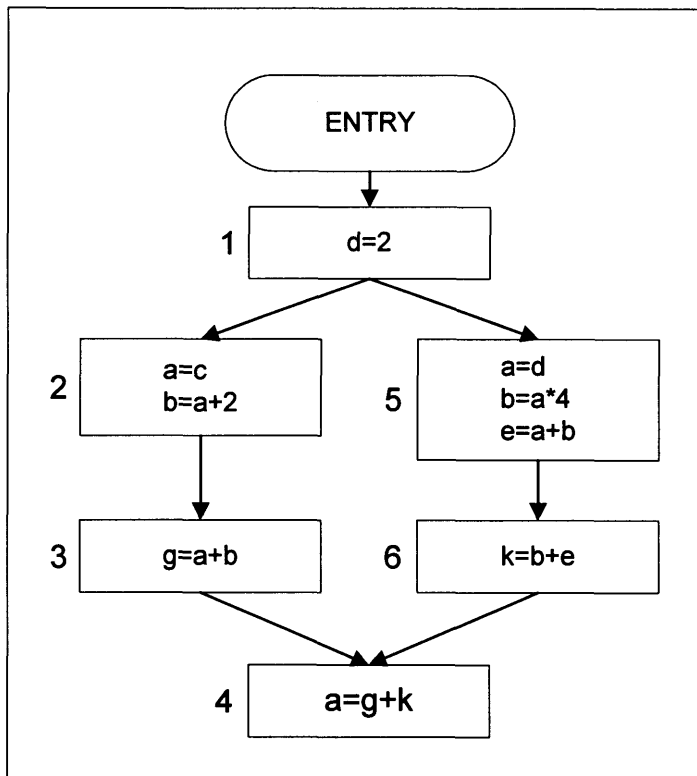
```

PROC Supertrace_Pass(BASIC_BLOCK)
  CALL Supertrace_Pass2(BASIC_BLOCK) // see FIGURE 5-2B
  SAVEREGS(*) = REGREF(*)
  for all SUCC(BASIC_BLOCK), S do
    if S != SUPERTRACE_HEAD(BASIC_BLOCK) ^
      SUPERTRACE_HEAD(S) == SUPERTRACE_HEAD(BASIC_BLOCK) then
      for J = 1 to numregs do
        REGREF[J] = SAVEREG[J]
        if REGREF[J] != ∅ then
          V1 = REGREF[J]
          if LIVEBEFORE(S,V1) then
            VARREF[REGREF[J]] = J
          else
            REGREF[J] = ∅
          fi
        fi
      rof
      CALL Supertrace_Pass(S)
    fi
  rof
END Supertrace_Pass

```

STATE SAVING ALGORITHM FOR FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-2C

We now present an example showing the operations performed by the forward supertrace register allocation algorithm. The CFG shown in Figure 5-3 has two supertraces. The first supertrace contains basic blocks one, two, three, five, and six. The second supertrace contains a single basic block, basic block four. Variables *a*, *b*, *d*, and *e* are supertrace local because they are not live coming in to the two supertrace heads (basic blocks one and four). Variables *c*, *g*, and *k* are supertrace global in our example. Variable *c* is live coming in to basic block two and because there is no definition in basic block one, it must be a global variable or parameter to a subroutine. Variables *g* and *k* are supertrace global because they are live coming into the supertrace head at basic block four. The local variable *d*, defined in basic block one is not live along the path from basic block one to basic block two. The variable *d* is live along the path from basic block one to basic block five. In the first assignment statement of basic block five, the variable *a* subsumes *d*.



EXAMPLE FLOW GRAPH FOR SUPERTRACE REGISTER ALLOCATION
FIGURE 5-3

Table 5-1 shows the initialization step of the forward register supertrace register allocation algorithm. Entries in the table show the LIVESTATE values of {live, dead} as 'L' and 'D' respectively. The LASTUSE values of {TRUE, FALSE} are 'T' and 'F' in the table. Note that LASTUSE is not important for variable definitions; we show this as an '_' character. The table shows the LIVESTATE and LASTUSE for all instructions in the basic block. Instructions in each basic block are in reverse order, the same order used by the algorithm. During initialization we compute the LIVESTATE and LASTUSE for local and global variables. The subsequent allocation pass deals with local variables only.

TABLE 5-1 FORWARD PASS SUPERTRACE Register Allocation Initialization								
Block	Instr.	LIVESTATE/LASTUSE						
		LOCAL				GLOBAL		
		a	b	d	e	c	g	k
1	=d			D/				
2	=a,b=	L/F	D/					
	=c,a=	D/				D/F		
3	=a,b,g=	L/T	L/T				D/	
4	=g,k,a=	D/					D/F	D/F
	dead							
5	=a,b,e=	L/T	L/F		D/			
	=a,b=	L/F	D/					
	=d,a=	D/		L/T				
6	=b,e,k=		L/T		L/T			D/

Table 5-2 shows the register allocation for our example. Supertrace register allocation saves the register state and permits LIVEAFTER variables to pass through to successors that are members of the same supertrace.

TABLE 5-2 FORWARD PASS SUPERTRACE REGISTER ASSIGNMENT							
Block	Instruction	REGREF				VARREF	
		a	b	d	e	R1	R2
1	d= R1=			R1		d	
2	=c,a= =c,R1= =a,b= =R1,R2=	R1				a	
			R2				b
3	=a,b,g= =R1,R2,g=	∅	∅			∅	∅
5	=d,a= =R1,R1= =a,b= =R1,R2= =a,b,e= =R1,R2,R1=	R1		∅		a	
			R2				b
		∅			R1	e	
6	=b,e,k= =R2,R1,k=		∅		∅	∅	∅
4	dead						
	dead						

The first table column shows the basic block numbers assigned during the depth first traversal of the CFG. The table also shows the original instructions and the register replacements that occur during register allocation and supertrace traversal. Global data constructs, REGREF and VARREF, show the assignment and clearing of registers and variables. We use the symbol ' \emptyset ' to represent clearing a value in REGREF or VARREF. The sequence of basic blocks in the table reflect the depth first sequence used in traversing the supertrace.

Statements from the input program and results from forward pass register assignment is in TABLE 5-3. Rows in the table show basic blocks in DFST order, statements are in first to last order.

TABLE 5-3 FORWARD PASS SUPERTRACE REGISTER ALLOCATION RESULTS		
Block	Original Program	Forward Supertrace Results
1	d=2	R1=2
2	a=c b=a+2	R1=c R2=R1+2
3	g=a+b	g=R1+R2
5	a=d b=a*4 e=a+b	R1=R1 R2=R1*4 R1=R1+R2
6	k=b+e	k=R2+R1
4	a=g+k	<dead code, instruction removed>

Optimality Criteria

Our algorithm generates an optimal register allocation in that no allocation can use fewer registers. We demonstrate this by casting the register allocation problem into a graph coloring framework. The CFG of a supertrace is a tree, and each variable live range is a subtree of the supertrace. We construct an undirected register interference graph where each node represents a live range. An edge connects two nodes if the live ranges intersect. It is well known that the interference graph of subtrees of a tree forms a *triangulated* or *chordal* graph[11]. The allocation of registers to live ranges corresponds directly to a coloring of the interference graph, where adjacent nodes must have different colors.

Any interference graph coloring must use at least as many colors as the size of the largest clique (a completely connected subset of nodes). Our algorithm requires a new register only when all the previously used registers are already allocated to variables at some point in the supertrace. The live ranges needing the new register interferes with all the live ranges currently allocated registers, and so all the variables at any point in the supertrace form a clique. Thus, our algorithm finds a coloring that uses exactly the number of colors in the largest clique, and so is optimal.

Variable Renaming in Modified Supertrace Algorithm

Modifying the supertrace register allocation algorithm to use a bottom up traversal of the supertrace eliminates the need to save register state between basic blocks. We need variable renaming in this algorithm to preserve dependence relations of instructions between basic blocks. Our motivation in creating this algorithm is to reduce the number of iterations over basic blocks and their instructions. A feature of this algorithm is that it does not save the register state between basic blocks in a supertrace. It is possible to implement variable renaming as part of the modified supertrace algorithm. We chose to implement this algorithm separately because of expedience and data collection requirements for our research.

The following algorithm renames local variables in supertraces that otherwise would cause incorrect register allocation in the modified supertrace register allocation algorithm. Unlike the forward pass supertrace algorithm, we do not save the state of register assignments in the modified supertrace algorithm. Definitions of a variable in two or more branches of the supertrace require that we must rename the variable. Renaming the variable removes the possibility that register values differ between branches of the supertrace. Renaming variables splits the live range of the original variable.

The algorithm to rename variables seeks the definitions of supertrace local variables. Starting at a basic block defined as a supertrace head and proceeding through the successors we build a list of variable definitions. If a basic block has only

one successor there is no need to rename variables in the basic block or its successor because variables live across this basic block boundary do not change. When a basic block has more than one successor, it is necessary to determine if there are any redefinitions of variables from our list in either successor subtree. The algorithm renames variables from our list if definitions exist for the variable in a successor's subtree. It is not necessary to rename all instances of the variable, as this would compound the problem we are trying to solve. We find the last definition in the instructions of the basic block and rename it. The algorithm renames subsequent uses of the variable found in later instructions of the basic block. In some cases, we must traverse through the basic block's predecessors to find the last definition and begin the renaming at that point in the supertrace. For each successor subtree we rename all uses of the variable up to the first definition. In summary, we rename variables from the last definition and subsequent uses in the basic block up to (but not including) the first definition of the variable in each of the successor subtrees. We show the main function in Figure 5-4.

```

PROC St_Var_Rename(BLOCK, STHEAD)
  for instructions, I in BLOCK do
    LIST = LIST + TARGET(I)
  rof

  if BLOCK has more than 1 successor then
    for LIST, J=1,N do
      if definitions exist in either successor chain for LIST[J] then
        CALL Rename_LastDef_Fwd(BLOCK, STHEAD, LIST[J], NEW_VAR_NAME)
        for succ(BLOCK), SBLOCK do
          CALL Rename_Upto_FirstDef(SBLOCK, STHEAD, LIST[J], NEW_VAR_NAME)
        rof
      fi
    rof
  fi

  for all SUCC(BLOCK), SBLOCK do
    if Supertrace_Child(SBLOCK, STHEAD) then
      CALL St_Var_Rename(BLOCK, STHEAD)
    fi
  rof
end St_Var_Rename

```

DRIVER ALGORITHM FOR VARIABLE RENAMING

FIGURE 5-4

Clearly, if a supertrace local variable is live out of a basic block and live coming into two or more successors, the variable represented by a register must have the same value and reflect the original definition. The problem we address with renaming arises when definitions in one branch of the supertrace overwrite the values used in the other branch. Note that the list of variable definitions, LIST, is initially NULL.

The functions shown in Figures 5-4A and 5-4B perform variable renaming by recursively visiting basic blocks within a supertrace. There are two 'directions' in renaming variables within a supertrace. The algorithm in Figure 5-4A renames a variable from the most recent definition through all subsequent uses. This form of variable renaming will traverse basic block predecessors to find the instruction where a variable definition occurs. The other 'direction' used to rename a variable involves following basic block successors until finding the variable definition in an instruction list. The algorithm in Figure 5-4B renames all uses of a variable up to the definition of the variable.

```

PROC Rename_LastDef_Fwd(BLOCK, STHEAD, VAR, NEW_VAR_NAME)
  FOUND=FALSE
  for instructions, I of BLOCK in reverse order and FOUND == FALSE do
    for operands, V1 of I do
      if V1 == VAR then
        V1 = NEW_VAR_NAME
      fi
    rof
    if Target(I) == VAR then
      Target(I) = NEW_VAR_NAME
      FOUND = TRUE
    fi
  rof
  if FOUND == FALSE and Supertrace_Child(PRED(BLOCK), STHEAD) then
    Call Rename_LastDef_Fwd(PRED(BLOCK), STHEAD, VAR, NEW_VAR_NAME)
  fi
end Rename_LastDef_Fwd

```

BACKWARD TRAVERSAL ALGORITHM FOR VARIABLE RENAMING
FIGURE 5-4A

```

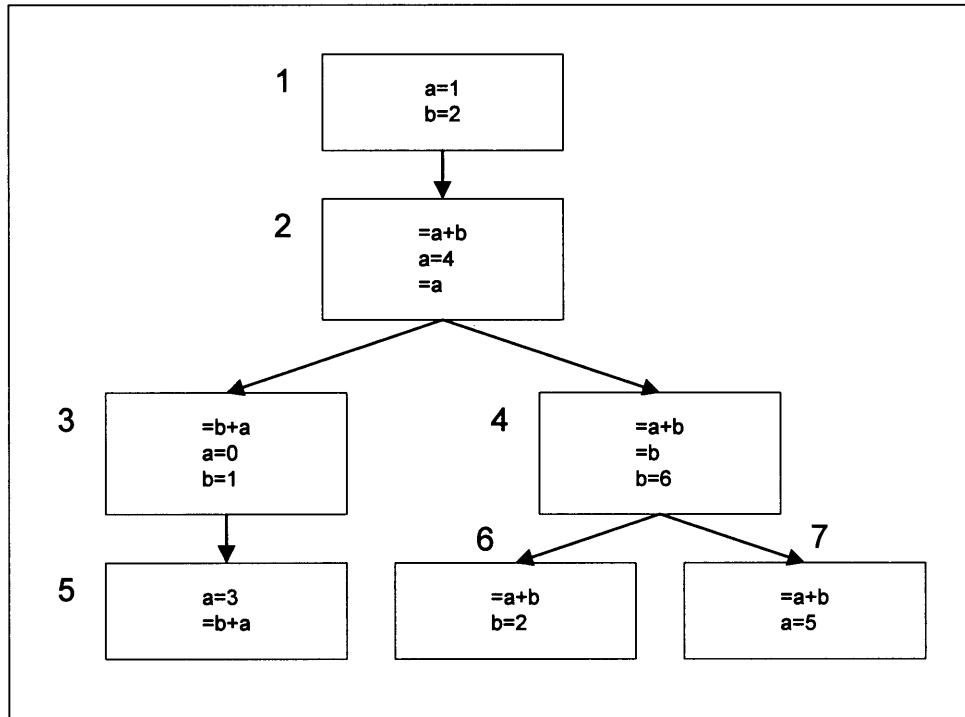
PROC Rename_Upto_FirstDef(BLOCK, STHEAD, VAR, NEW_VAR_NAME)
  FOUND = FALSE
  for instructions, I of BLOCK and FOUND == FALSE do
    for operands, V1 of I do
      if V1 == VAR then
        V1 = NEW_VAR_NAME
      fi
    rof
    if TARGET(I) == VAR then
      FOUND = TRUE
    fi
  rof
  if FOUND == FALSE then
    for succ(BLOCK), SBLOCK do
      if Supertrace_Child(SBLOCK, STHEAD) then
        CALL Rename_Upto_FirstDef(SBLOCK, STHEAD, VAR, NEW_VAR_NAME)
      fi
    rof
  fi
end Rename_Upto_FirstDef

```

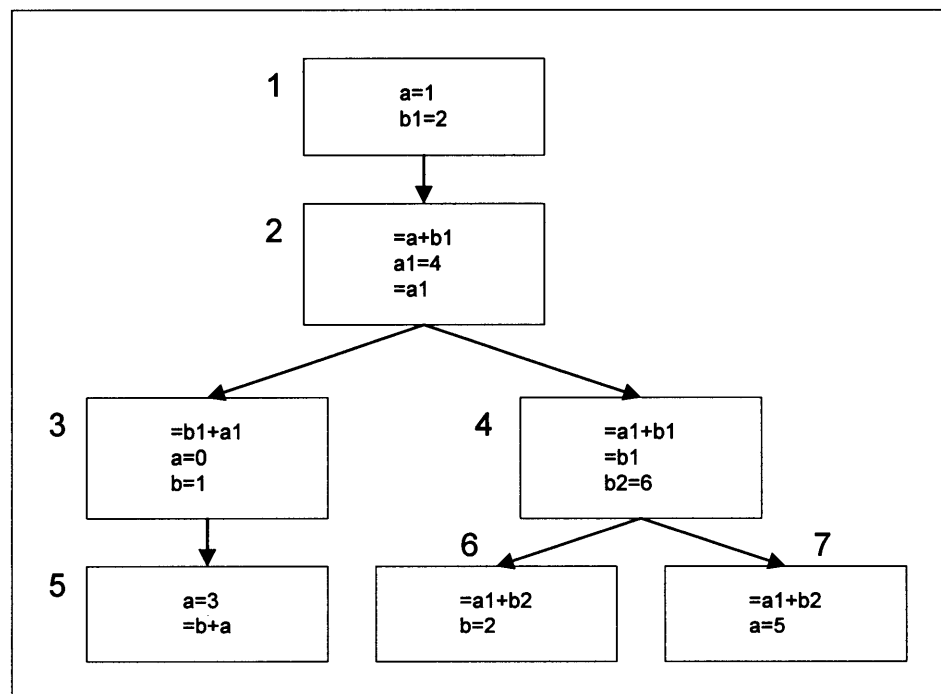
FORWARD TRAVERSAL ALGORITHM FOR VARIABLE RENAMING
FIGURE 5-4B

It is beneficial to perform dead code elimination before live range splitting. The assignment statement ($b=2$) in basic block six is dead because there are no subsequent uses of the variable 'b'. Because we did not perform dead code elimination before live range splitting it is necessary to rename the assignment in basic block four ($b=6$) and the use in basic block seven ($=a+b$). Performing live range splitting after dead code elimination has the potential of reducing the number of statements in the flow graph. The positive side effect of performing dead code elimination before variable renaming is reducing the number of variables renamed.

We present an example of variable renaming in Figures 5-5 and 5-6. Figure 5-5 shows the original CFG; Figure 5-6 shows the CFG after renaming.



EXAMPLE FLOW GRAPH BEFORE VARIABLE RENAMING
FIGURE 5-5



EXAMPLE FLOW GRAPH AFTER VARIABLE RENAMING
FIGURE 5-6

Modified Supertrace Algorithm

The entry point for the modified supertrace algorithm is in Figure 5-7. Initialization sets the LIVESTATE of all variables to dead. Using this algorithm requires renaming variables defined on two or more branches. The second pass, shown in Figure 5-7a, calculates the live ranges and performs dead code elimination. Variable uses local to the basic block with a dead LIVESTATE result in the LIVESTATE being set to live and LASTUSE being set to TRUE. Dead code elimination removes local definitions from the instruction list having a LIVESTATE of dead. The register allocation step, shown in Figure 5-7b, assigns registers for uses and definitions that are local to the supertrace. The register allocation pass requires processing basic blocks in the opposite order of the first pass. This algorithm has the effect of treating all basic blocks in a supertrace as one 'congealed' basic block.

```

PROC ForwardModPass(GRAPH)
  for all basic blocks, B of flowgraph, GRAPH do
    if SupertraceHead(B) then
      for all variables, V1 in SYMBOL_TABLE do
        LiveState(V1) = dead
      rof
      CALL ST_Var_Rename(B, B) // see FIGURE 5-4
      CALL Mark(B)
      CALL Alloc(B)
    fi
  rof
END ForwardModPass()

```

DRIVER ALGORITHM FOR MODIFIED FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-7

Before computing the live ranges, we set the LIVESTATE of all variables to *dead*. It is not important that global variables have any attribute values at this point. We compute the live range and then assign registers for supertrace local variables for each supertrace encountered in the CFG. Because every basic block in the CFG is a supertrace or a member of a larger supertrace, we know the algorithm will visit every basic block in CFG.

We present the live range function *Mark()* in Figure 5-7A. It visits each basic block in the supertrace in a bottom up depth first traversal. It is necessary to use a

bottom-up traversal of nodes in the supertrace to maintain the LIVESTATE information for variables. We process instructions from last to first.

```

PROC Mark(BASIC_BLOCK, STHEAD)
  for all SUCC(BASIC_BLOCK), SBLOCK do
    if SupertraceChild(SBLOCK, STHEAD) then
      CALL Mark(SBLOCK, STHEAD)
    fi
  rof
  for all instructions, I of BASIC_BLOCK in reverse order do
    for all variables, V1 of I do
      LASTUSE(I,V1) = FALSE
    rof
    V1 = TARGET(I)
    if (V1) ^ STATE(V1) == local then
      if LIVESTATE(V1) == dead then
        CALL REMOVE_INSTR(I,V1)
      else
        LIVESTATE(V1) = dead
      fi
    fi
    for all operands, V2 of instruction I do
      if STATE(V2) == local ^ LIVESTATE(V2) == dead then
        LASTUSE(I,V2) = TRUE
        LIVESTATE(V2) = live
      fi
    rof
  rof
END Mark

```

INITIALIZATION ALGORITHM FOR MODIFIED FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-7A

We present the register allocation function in Figure 5-7B. Register allocation and assignment for this algorithm is complicated by the need to traverse basic blocks in the supertrace in exactly the opposite order used to compute variable live ranges. We perform a top-down preorder left to right traversal of the supertrace. The sequence we use for allocating instructions in basic blocks is necessary to maintain valid register assignments across basic blocks.

```

PROC Alloc(BASIC_BLOCK)
  for all instructions, I of BASIC_BLOCK do
    for all operands, V1 of I do
      if STATE(V1) == local then
        CALL AllocReg(V1)
        if LASTUSE(I,V1) = TRUE then
          CALL FreeReg(WhichReg(V1))
        fi
      fi
    rof
    V2 = TARGET(I)
    if (V2) ^ STATE(V2) == local then
      CALL AllocReg(V2)
    fi
  rof
  for all SUCC(BASIC_BLOCK), SBLOCK in reverse order do
    if Supertrace_Child(SBLOCK, BASIC_BLOCK) then
      CALL Alloc(SBLOCK)
    fi
  rof
end Alloc

```

ALLOCATION ALGORITHM FOR MODIFIED FORWARD SUPERTRACE REGISTER ALLOCATION
FIGURE 5-7B

Using the CFG from Figure 5-6, we show the results of register allocation with the modified supertrace algorithm in Table 5-4 and Table 5-5. We show the states *LIVESTATE* $\{live, dead\}$ and *LASTUSE* $\{TRUE, FALSE\}$ as $\{L,D\}\{T,F\}$ along with the basic block number and instruction. The algorithm traverses basic blocks in the supertrace using a depth first left to right traversal. Processing statements or instructions within each basic block from last to first order maintains data dependence relations. Table 5-4 shows the initial state of each variable and the results after live range calculations. We use the word 'dead' in the following tables when referring to dead code removed from the flow graph by the allocation algorithm.

TABLE 5-4 MODIFIED SUPERTRACE REGISTER ALLOCATION INITIALIZATION						
block	instr	LIVESTATE/LASTUSE				
		a	a1	b	b1	b2
5	=b+a	L/T		L/T		
	a=3	D/F				
3	b=1			D/F		
	a=0	dead				
	=b1+a1		L/T		L/T	
6	b=2		L/F	dead		
	=a1+b2					L/T
7	a=5	dead				
	=a1+b2		L/F			L/F
4	b2=6					D/F
	=b1				L/F	
	=a1+b1		L/F		L/F	
2	=a1		L/F			
	a1=4		D/F			
	=a+b1	L/T			L/F	
1	b1=2				D/F	
	a=1	D/F				

Table 5-5 presents the register assignment for our example. Traversal of the basic blocks in the register allocation pass must be in the exact *opposite* order used during variable live ranges calculations. We use a top down preorder traversal to accomplish the register assignment traversal of the supertrace. Register assignment processes instructions within each basic block in order, from first to last. We show instructions before and after local register assignment. Register assignment for local variables is in Table 5-5. Scanning down the columns beneath the REGISTER ASSIGNMENT title permits finding register live ranges that coincide. There are three coinciding register live ranges in the supertrace at basic block four after the third instruction (b2=6). We show the return of a register to the free register pool with the symbol, 'Ø'.

We show the results from the modified forward pass supertrace register allocation algorithm in Table 5-6, comparing the original instructions to the output. The word 'dead' replaces instructions removed from the flow graph.

TABLE 5-5 MODIFIED SUPERTRACE REGISTER ASSIGNMENT							
block	instr	alloc	REGISTER ASSIGNMENT				
			a	a1	b	b1	b2
1	a=1	R1=1	R1				
	b1=2	R2=2				R2	
2	=a+b1	=R1+R2	∅			R2	
	a1=4	R1=4		R1			
	=a1	=R1		R1			
4	=a1+b1	=R1+R2		R1		R2	
	=b1	=R2				R2	
	b2=6	R3=6					R3
7	=a1+b1	=R1+R2		R1		R2	
	a=5		dead				
6	=a1+b2	=R1+R3		R1			∅
	b=2				dead		
3	=b1+a1	=R2+R1		∅		∅	
	a=0		dead				
	b=1	R1=1			R1		
5	a=3	R2=3	R2				
	R2=3	R2=3					
	=b+a	=R1+R2	∅		∅		

TABLE 5-6 MODIFIED SUPERTRACE REGISTER ALLOCATION RESULTS		
block	original instruction	Modified Supertrace Results
1	a=1	R1=1
	b1=2	R2=2
2	=a+b1	=R1+R2
	a1=4	R1=4
	=a1	=R1
4	=a1+b1	=R1+R2
	=b1	=R2
	b2=6	R3=6
7	=a1+b2	=R1+R2
	a=5	dead
6	=a1+b2	=R1+R3
	b=2	dead
3	=b1+a1	=R2+R1
	a=0	dead
	b=1	R1=1
5	a=3	R2=3
	=b+a	=R1+R2

Conclusion

The supertrace register allocation algorithm is a forward pass basic block register allocation algorithm. Modifications in the supertrace algorithm permit variable live ranges to span basic block boundaries. The supertrace register allocation algorithm is similar to interval graph algorithms[10], operating in polynomial time. Solutions for register allocation without spilling are optimal, given the fact supertrace register allocation is a modified version of the basic block forward pass register allocation algorithm. Register allocation involving spill code provides a non-optimal solution. The forward supertrace algorithm makes two passes through the basic blocks; once during initialization and once during register allocation. The algorithm visits instructions within each basic block twice; once during initialization and once during register allocation. The algorithm incurs the cost of saving and restoring register state once for every basic block assigned to a supertrace.

The modified supertrace algorithm visits all basic blocks in a CFG three times. When the algorithm finds a basic block that is a supertrace head initialization of all variables in the symbol table occurs before live range computations or register assignment. The algorithm visits instructions within each basic block twice; once during live range calculations and once during register assignment. The requirement for variable renaming creates an additional cost of using our nonlinear algorithm to perform live range splitting. Performing variable renaming is possible with a less costly algorithm or by integrating live range splitting directly into the modified supertrace algorithm.

CHAPTER 6

CREATING LARGER SUPERTRACES

Introduction

This chapter presents two mechanisms to create larger supertraces; basic block cloning and loop unrolling. The supertrace register allocation algorithms depend on trees of basic blocks to allocate local variables. By modifying the structure of a flow graph with basic block cloning, register lifetimes for supertrace local variables can span more basic blocks. Increasing the depth of supertraces by cloning flow graph merge points reduces the number of supertraces present in a given flow graph. Cloning basic blocks during the creation of a supertrace increases the number of variables available for local register allocation.

Without cloning, supertrace branches end at the point where basic block successor chains join. Cloning basic blocks with more than one predecessor permits the supertrace to grow larger and include more basic blocks. The following discussion of basic block cloning is in the context of creating larger supertraces for the supertrace register allocation algorithms. Basic block cloning has no benefit for basic block register allocation. We examine two different flow graph traversals, depth first and breadth first, to create supertraces. Descriptions of several different limits are in the following section that affect the size and form of supertraces. Loop back edges form barriers that limit supertrace growth.

Loop unrolling increases the number of basic blocks by copying the flow graph structure that exists between the start and end of a loop. Loop unrolling by itself offers

no direct benefit toward increasing larger trees of basic blocks. Combining loop unrolling with basic block cloning increases the number of basic blocks in a CFG dramatically. Because the number of basic blocks can increase so quickly, we impose limits to control the growth rate of the flow graph.

Limits

We use limits in the basic block cloning algorithm and the loop unrolling algorithm to establish upper limits on the number of basic blocks in a supertrace or loop, respectively. Both optimization techniques increase the number of basic blocks in the CFG. Using limits to control the number of basic blocks added to the CFG balances code size against the number of variables allocated locally by the supertrace register allocation algorithms.

Basic block cloning permits the application of several different limits. Creating supertraces with the maximum number of basic blocks is possible using basic block cloning. Loop structures create boundaries that supertraces can not cross. Creating a supertrace without limits, by depth first or breadth first traversals generates identical results. We may choose to establish a limit for the total number of basic blocks in a supertrace, *block_count_limit*. Adding basic blocks to the supertrace continues until the number of basic blocks approaches the *block_count_limit*. The second limit we can impose on supertrace formation is the *maximum_depth* of a supertrace, calculated by the number of successor links between the supertrace head and any supertrace member. The implication of using a *block_count_limit* versus *maximum_depth* is dependent on the structure of the flow graph. To obtain the maximum number of supertrace local variables for a given flow graph, the source program requires careful examination. Limits imposed on supertrace size, the number of basic blocks, and the method of traversing the supertrace have an impact on the number of supertrace local variables available for local register allocation.

The first loop unrolling limit we examine is unrolling all loops once. The number of instructions in each loop is: $loop_{instructs} = loop_{instructs} * 2$. In the case of nested loops,

where $loop[0]$ is the innermost loop, and $loop[n]$ is the outermost loop, our equation for the number of instructions at the outermost loop after unrolling each loop once is:

$$loop[n]_{instruct} = loop[n]_{instruct} * 2 + loop[n-1]_{instruct} * 2 + \dots + loop[0]_{instruct} * 2$$

Setting a limit on the number of iterations to unroll has the effect of increasing the number of instructions by: $loop_{instructs} = loop_{instructs} * unroll_{count}$. This unrolling technique is the most arbitrary of the limits we examine. Unrolling loops with this limit causes a loop containing few instructions to be unrolled the same number of times as loops containing hundreds or thousands of instructions. The use of this technique requires careful characterization of the input program to insure applicability.

It is possible to create an upper limit on the number of instructions contained in a loop. Setting a limit on the number of instructions contained in a loop creates uniformly sized loops. This technique requires counting the number of instructions in a loop and unrolling the loop until the number of instructions approaches $\lceil loop_{instruct} \rceil$. In the case of nested loops, the inner loop unrolls up to the $loop_{instruct}$ limit, outer loops remain unchanged.

Unrolling loops limited by the number of basic blocks is another option. Loop unrolling in this fashion limits the replication of control flow constructs. Given our definition of a basic block, if a loop contains more than one basic block there must be a control flow construct or function call within the loop. This technique of loop unrolling is a benefit to the supertrace register allocation algorithms when combined with basic block cloning. We introduce basic block cloning in the following section. This technique counts the number of basic blocks, $block_{count}$, within each loop. The loop is unrolled until $block_{count} * 2$ approaches $\lceil block_{limit} \rceil$.

Basic Block Cloning

Cloning is the duplication of basic blocks in a flow graph that have more than one predecessor. This process is similar to *tail duplication*[15]. Each predecessor branches to a copy of the cloned basic block. An example of a basic block with more than one predecessor is the basic block at the confluence point for an “if-then-else”

statement. This basic block has two predecessors; one from the “then” branch, and one from the “else” branch. By cloning the basic block where the conditional branches merge, we can expand the flow graph without changing the flow of control or dependence relationships existing in the original flow graph. Basic block cloning creates larger trees of basic blocks. This technique is beneficial to the supertrace register allocation algorithms.

Depth first creation of supertraces creates ‘long’ chains of basic blocks. This traversal method visits basic blocks following successor links, starting at the supertrace head and terminating at a limit threshold, or upon encountering a basic block containing a successor link that is loop back edge.

Breadth first traversal uses a left to right inorder sequence to build supertraces. Supertraces constructed with breadth first traversals create ‘wide’ supertraces. Limiting the size of the supertrace by the number of basic blocks or maximum depth cause the number of supertraces in the CFG to increase.

Cloning a basic block is necessary under two circumstances. If a basic block is already in the current supertrace and there is another successor link to the basic block we need to clone the basic block. The other condition where basic block cloning is necessary occurs when a basic block is already a member of another supertrace. Cloning basic blocks under the second condition removes the possibility of branching into the middle of a supertrace.

The algorithm presented in Figure 6-5 is the entry point for supertrace formation. We pass a flow graph to the function and set LIMIT_CONDITION to one of the defined limits. This function follows lexical links in the flow graph and enqueues basic blocks identified as supertrace heads. The inner while loop builds the supertrace by dequeuing a basic block and then calling Build_Supertrace(). Cloning basic blocks in Build_Supertrace() requires that we rebuild the flow graph spanning tree and loop bodies to update internal data structures.

```

PROC Supertrace(GRAPH)
  CALL INIT_QUEUE()
  LIMIT_CONDITION = maximum_size | block_count_limit | maximum_depth
  NODE = flow graph entry
  while NODE != NULL do
    if NODE is a supertrace head then
      SBHEAD(NODE)=NODE
      ENQUEUE_HEAD(NODE)
      while queue is not empty do
        TNODE = DEQUEUE_HEAD()
        if LIMIT_CONDITION AND Build_Supertrace(GRAPH,TNODE) then
          rebuild spanning tree for flow graph
          rebuild loops in flow graph
        fi
      elihw
    fi
    NODE = GG_NEXT(NODE)
  elihw
END Supertrace

```

DRIVER ALGORITHM FOR BASIC BLOCK CLONING
FIGURE 6-5

The function `Build_Supertrace()`, presented in Figure 6-5A, follows the successor links from `NODE`, adding basic blocks to the supertrace. Successors to the basic block that are spanning tree ancestors are targets of back edges or targets of an exit edge. If the successor of `NODE` is not a member of the same loop, the successor is a member of another supertrace.

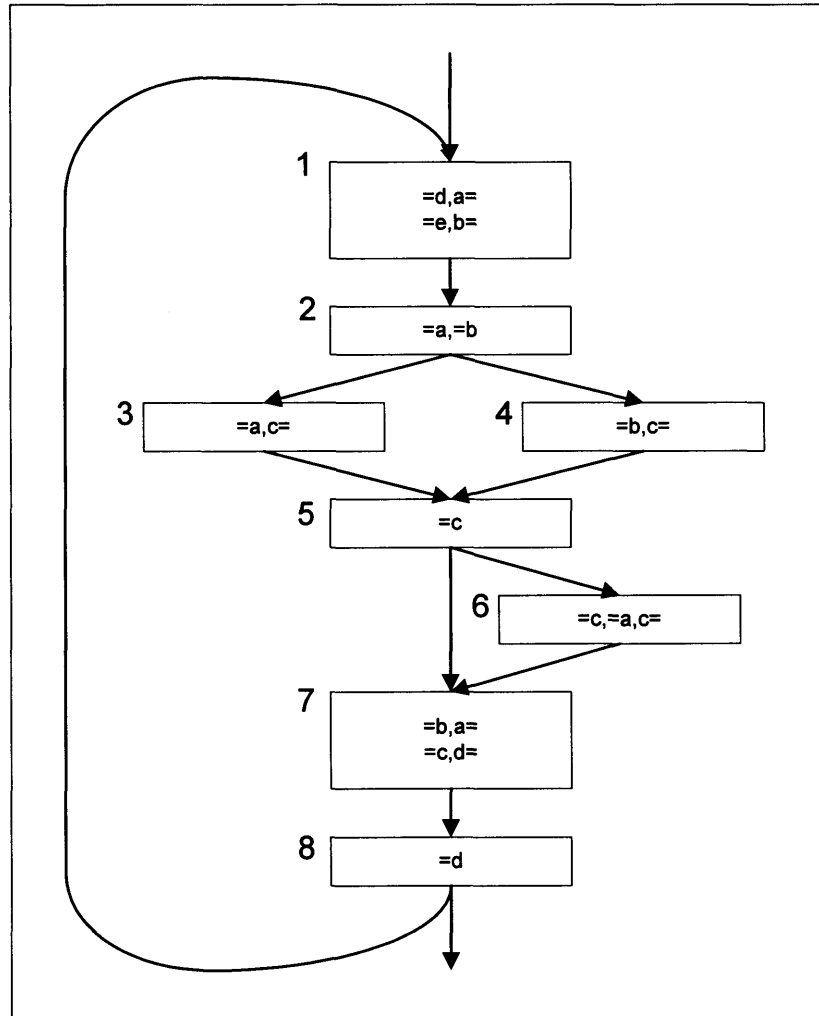
```

PROC Build_Supertrace(GRAPH,NODE)
for all SUCC(NODE), S of basic block, NODE do
  if S is spanning tree ancestor then
    do nothing; S is target of backedge and not member of this supertrace
  elseif S in supertrace & not assigned yet
    if cloning is permitted  $\vee$  PRED_COUNT(S) == 1 then
      if S is a member of the same loop as NODE then
        SBHEAD(S) = SBHEAD(NODE)
      if depth-first then
        ENQUEUE_HEAD(S)
      elseif breadth-first then
        ENQUEUE_TAIL(S)
      fi
    fi
  elseif basic block already in supertrace
    NEWNODE = COPYNODE(S)
    SBHEAD(NEWNODE) = SBHEAD(NODE)
  else basic block must be in different supertrace
    if cloning is permitted  $\wedge$  NODE is not a supertrace head then
      NEWNODE = COPYNODE(S)
      SBHEAD(NEWNODE) = SBHEAD(NODE)
    fi
  fi
  if NEWNODE was created then
    establish lexical links
    for all SUCC(S), SS do
      establish flowgraph links from NEWNODE to SS
    rof
    add flowgraph link from NODE to NEWNODE
    remove flowgraph links from NODE to S
    if depth-first then
      ENQUEUE_HEAD(NEWNODE)
    elseif breadth-first then
      ENQUEUE_TAIL(NEWNODE)
    fi
  fi
rof
if NEWNODE was created then RETURN TRUE
else RETURN FALSE
END Build_Supertrace

```

TRAVERSAL/GENERATION ALGORITHM BASIC BLOCK CLONING
FIGURE 6-5A

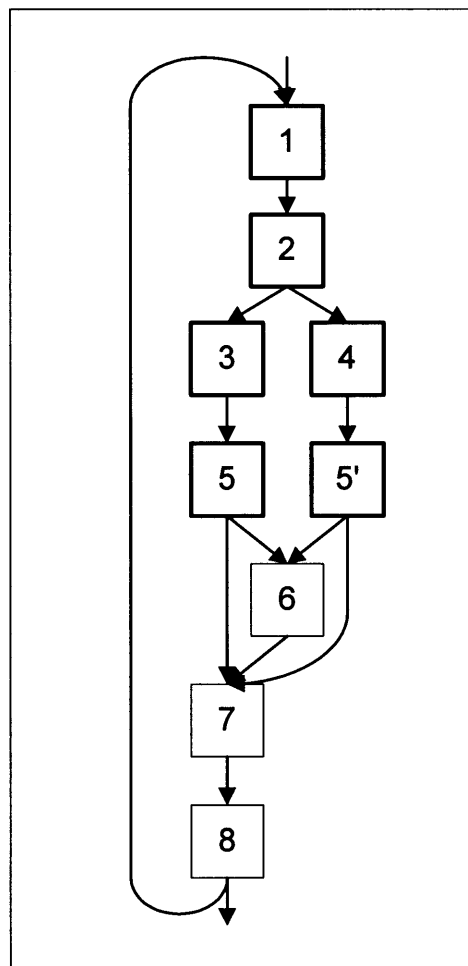
We now present an example of creating a supertrace with unlimited basic block cloning using a breadth-first traversal of the supertrace. The flow graph in Figure 6-6 shows a CFG with an if-then-else construct followed by an if-then construct inside a loop. The supertrace head is the basic block labeled '1'. Variable uses and defs are inside the basic blocks, one line per instruction.



EXAMPLE FLOW GRAPH BEFORE BASIC BLOCK CLONING
FIGURE 6-6

Two functions generate supertraces with basic block cloning. The function `Supertrace_Pass()` initializes each supertrace and calls `Build_Supertrace()` until exhausting the queue of potential supertrace members or reaching an imposed limit. The `Build_Supertrace()` function is responsible for processing the successors of the basic block, adding legal members or cloning and adding basic blocks already in the supertrace. `Build_Supertrace()` performs basic block cloning and adjusts successor and predecessor lists to insert cloned basic blocks into the flow graph. We start the example in `Supertrace_Pass()` where basic block 1 is a supertrace head. We call `Build_Supertrace()` passing basic block 1 as an argument. In `Build_Supertrace()` we add basic block 2 into the supertrace and enqueue it so that its successors may follow during the breadth-first traversal. `Supertrace_Pass()` dequeues basic block 2 and calls

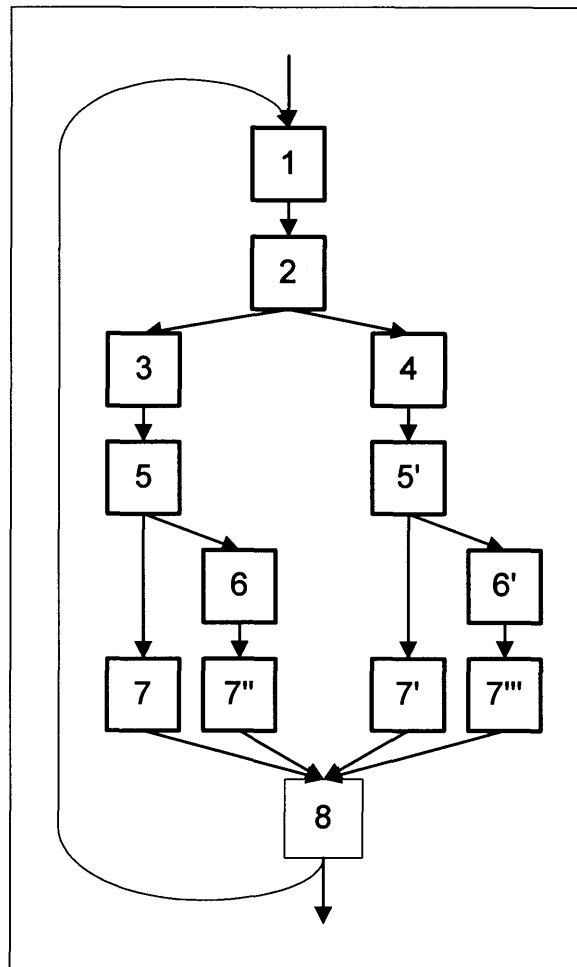
Build_Supertrace(). Build_Supertrace() adds basic blocks 3 and 4 to the supertrace and enqueues them. Supertrace_Pass() dequeues basic block 3 and calls Build_Supertrace(). Basic block 3 has only one successor, basic block 5. At this point we add basic block 5 to the supertrace and enqueue it. Build_Supertrace dequeues basic block 4 and passes this as an argument to Build_Supertrace(). The only successor of basic block 4 is basic block 5, but it is already in the supertrace. We clone basic block 5 creating another copy of this basic block, calling it 5'. Modifications to the successor and predecessor lists permit basic block 4 to point to basic block 5'. The algorithm assigns basic block 5' to the same successors as basic block 5, enqueue basic block 5' and returns to Supertrace_Pass(). We show the supertrace at this point in its creation in Figure 6-7.



EXAMPLE FLOW GRAPH DURING BASIC BLOCK CLONING
FIGURE 6-7

We dequeue basic block 5 and call Build_Supertrace(). We add Basic blocks 6 and 7 to the supertrace and enqueue these basic blocks. Basic block 5' is dequeued

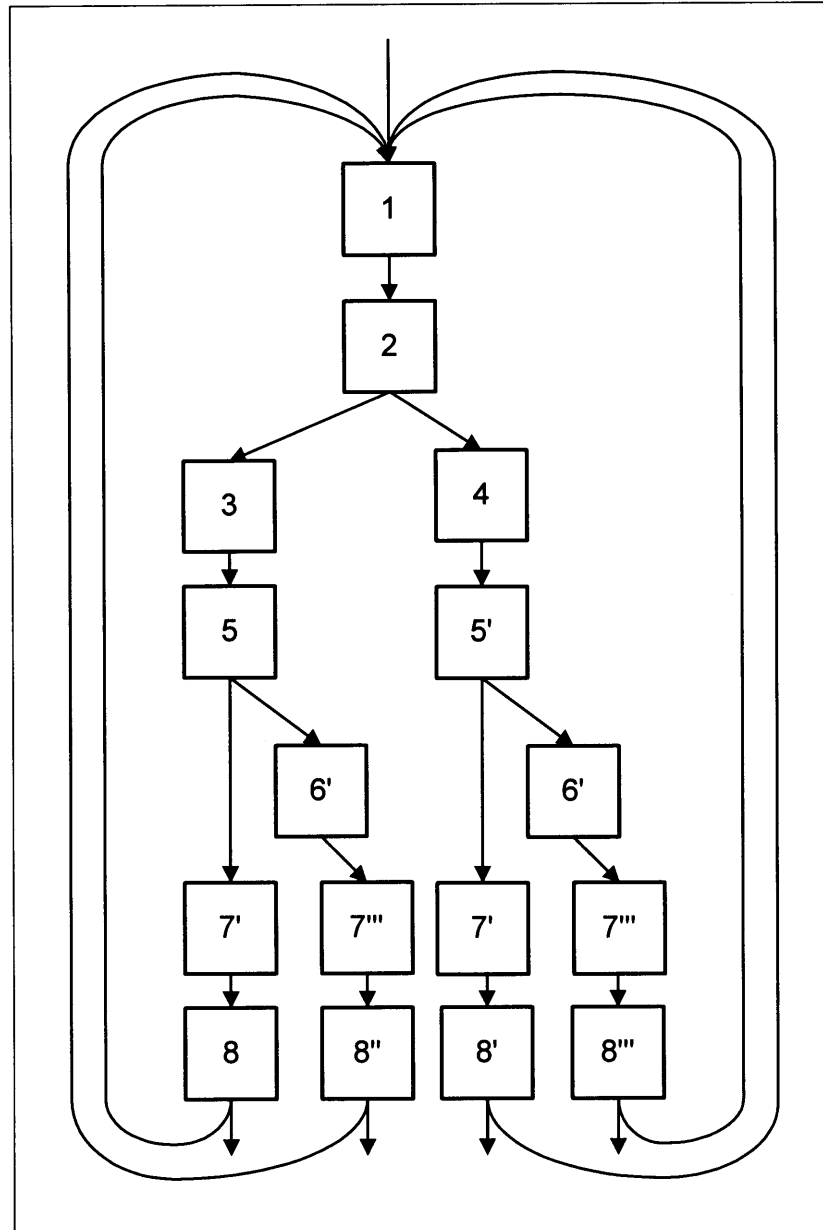
and sent to `Build_Supertrace()`. Basic block 6 clones to 6', basic block 7 clones to 7'. Basic block 5' successors are now basic blocks 6' and 7'. Basic block 6 is dequeued and sent to `Build_Supertrace()`. We need to clone basic block 7 to permit basic blocks 5 and 6 to have unique copies. Cloning basic block 7 creates basic block 7'', added to the queue after updating successor and predecessor lists, and flow graph links. The queue contains basic blocks 7, 6', 7', 7''. Basic block 7 is dequeued and processed by `Build_Supertrace()`. We add basic block 8 to the supertrace and enqueue it. Dequeuing and processing basic block 6' causes the creation of basic block 7'''. We add basic block 7''' to the supertrace, enqueue it, and the necessary updating of lists and links takes place. We show the supertrace at this point of its creation in Figure 6-8.



EXAMPLE FLOW GRAPH DURING BASIC BLOCK CLONING
FIGURE 6-8

The final sequence of dequeuing basic blocks and cloning creates a unique copy of basic block 8 for basic blocks 7', 7'', and 7'''. The difference between this final

sequence and the add/clone/enqueue sequence is when Build_Supertrace() processes basic blocks 8, 8', 8'', and 8'''. The Build_Supertrace() function performs no action because the only successor for basic block 8 is the supertrace head. We must consider the back edge of loops as an absolute supertrace limit. The complete supertrace is in Figure 6-9.



EXAMPLE FLOW GRAPH AFTER BASIC BLOCK CLONING
FIGURE 6-9

Loop Unrolling

Loop unrolling is possible at the source level, in the intermediate form, or at the machine code level. Unrolling loops in flow graphs creates a duplicate set of basic blocks. Each set of basic blocks has an induction variable update and a branch between the copy of the loop exit and the loop exit. Source level unrolling makes provisions so that a single induction variable update occurs. Production compilers often introduce preamble or postamble code to increase processor pipeline efficiency in loops known to execute an odd number of times.

Loops in a flow graph are bound by a loop head and a back edge. Loops may contain multiple basic blocks arranged in an arbitrary order. Unrolling loops has the side effect of increasing the number of forward edges in flow graphs because the cloned loop tail must have a flow graph edge to the successor(s) of the original loop tail. This optimization has no positive effect on local register allocation.

We show the loop unrolling algorithm in Figure 6-1. The process of unrolling loops requires the identification of *loop heads*, *loop tails*, and *back edges*. If a basic block is a target of a flow graph edge from a successor, the basic block is a *loop head*. The edge connecting the flow graph successor to the loop head is a *back edge*. The basic block originating the back edge is a *loop tail*. Loop unrolling works from the most deeply nested loop outward. Input to this optimization is the flow graph of a subroutine. Loop heads are basic blocks targeted by a back edge. The algorithm creates clones of original loop members. Lexical and flow graph connections establish the original loop as a template. Loop unrolling is complete after inserting the copy of the loop into the flow graph before the original loop. Changing loop back edges, adding an exit edge, and modifying the loop entry edges completes the task.

```

PROC Unroll(flow graph)
  build reverse spanning tree for flow graph
  annotate basic blocks making up loop heads, loop tails & loop membership
  for each loop, innermost to outermost do
    while selected limiting condition not met do
      clone the loop head and all nodes which are members of the loop
      add flow graph links between cloned nodes
      add flow graph links between cloned loop head and PRED(original loop head)
      add flow graph links between cloned loop tail and original loop head
      insert cloned loop before original loop head
      add exit edge between cloned loop tail and original loop tail
      update flow graph links to/from original loop head
      update flow graph links to/from original loop tail
    elihw
  rof
  rebuild reverse spanning tree for flow graph
  annotate basic blocks in flow graph for loop heads, loop tails & loop membership
END PROC Unroll

```

LOOP UNROLLING ALGORITHM
FIGURE 6-1

Conclusion

Loop unrolling and basic block cloning increase the number of basic blocks in flow graphs. Loop unrolling by itself does not increase the number of local variables or provide for the extension of local variable live ranges. Neither supertrace register allocation algorithms benefits from loop unrolling alone. Variable live ranges and the number of local variables do not change with loop unrolling. Supertrace formation with basic block cloning provides the supertrace register allocation algorithms with local variable live ranges that extend beyond basic block boundaries. We show how supertraces can form with basic block cloning that extend beyond the confluence points in flow graphs. We show how limits can guide basic block cloning within supertraces to prevent the number of basic blocks from increasing at a potentially geometric rate. The selection of breadth-first or depth-first traversal methods impacts the shape and number supertraces within a given flow graph. The use of supertrace register allocation with basic block cloning provides potential benefits and new insights into compiler optimization techniques.

CHAPTER 7 FUTURE WORK AND CONCLUSIONS

To validate this research, the next step should be experimental. An implementation of our algorithms should be compared to a global graph coloring register allocator, both with and without a separate basic block local allocator. Some of the data that should be collected and studied are listed here.

The first set of experiments would set the basis for our work. An experiment should be run to study how many variables the register allocation phase must deal with, and how many of these are basic block local variables. If only a small number of variables are basic block local, it is hard to justify a separate allocator for them, if only for maintenance reasons.

An experiment should be run to study the compile time of register allocation compared to the number of variables allocated. For a graph coloring global register allocator, we expect the compile time to grow faster than linearly with the number of variables. The time for a global allocator without a basic block local allocator should be compared to the time take when local variables are allocated by the linear algorithm. The hope is that removing the basic block local variables from the interference graph will allow the global allocator to run much faster.

The same experiment can be used to collect information about numbers of registers used and number of register spills added. The ratio of registers used for local variables and global variables should be collected, also. Since the basic block local allocator has perfect knowledge about subsequent uses of variables at spill points, it should translate into better choices of spilled variables.

The second set of experiments would study naturally occurring supertraces in the control flow graph. First, flow graphs should be studied to find supertraces, and data collected about their sizes. The first set of experiments should be repeated, replacing the basic block local allocator by the supertrace local allocator. If there are relatively few supertrace local variables, using the supertrace local allocator may give no compile time or spill performance benefits.

The third set of experiments would study the effects of unlimited basic block cloning. One experiment would study code growth, i.e., how much larger do programs get if a compiler does as much basic block cloning as it needs to find maximal supertraces. Our initial experiments along this line appear to support the claim that most programs grow by a factor of two or less, measured in basic blocks. The second set of experiments should be repeated on these maximal supertraces, to see the effect of cloning. With a larger program, compile time may again become an issue; the hope is that the faster supertrace register allocator on the larger program will allocate enough variables that the global coloring allocator on the rest of the variables will be quite fast.

The final set of experiments would study the effects of cloning limits, and depth first vs. breadth first supertrace creation. The limits are important only when unlimited cloning creates a program that is too large or takes too long to compile. With different limits used and supertrace construction order, the same program may give very different performance results. This set of experiments will study the sensitivity of the register allocation strategy to the cloning strategy.

In all the supertrace allocation experiments, the optimal forward allocator and nonoptimal faster backward allocator should be compared, to see whether there are significant performance benefits or costs for the backward allocator.

We did not study spill heuristics at all. Depending on how many variables are found to be supertrace local, spill heuristics may be either trivial or quite important. Some extension of the basic block local “most distant next use” spill heuristic may be appropriate and sufficient.

Future work may also study the idea of using the supertrace allocator to allocate global variables as well. This may require insertion of compensation or "fix-up" code when entering a new supertrace, if the source and target of a branch have allocated some globals to different registers. The advantages of using a fast, accurate allocator may outweigh the performance of a true global allocator that can only approximate control flow information.

Conclusion

We have extended the optimal basic block local register allocator to allocate registers to variables that are live across basic block boundaries, but are local to supertraces. We have shown two algorithms, one which is optimal in the sense that it uses the fewest registers possible, and another which may be more efficient. We have shown how to identify supertraces in control flow graphs. We have also shown algorithms to enlarge supertraces through basic block cloning. We hope that experimental evidence will support our claim this is a viable and useful alternative to using global graph coloring allocators.

BIBLIOGRAPHY

- [1] Aho, A.V., Sethi, R., Ullman, J.D., *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading Ma., 1986.
- [2] Bernstein, D., et. al., "Spill Code Minimization Techniques for Optimizing Compilers", ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 258-263, 1989.
- [3] Briggs, P., et. al., "Coloring Heuristics for Register Allocation", ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 275-284, 1989.
- [4] Callahan, D., Koblenz, B., "Register Allocation via Hierarchical Graph Coloring", Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pp.192-203, 1991.
- [5] Chaitin, G., et. al., "Register Allocation via Coloring", *Computer Languages*, Vol. 6, pp. 47-57, 1981.
- [6] Chaitin, G., "Register Allocation & Spilling Via Graph Coloring", Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, pp. 98-105, 1982.
- [7] Chow, F., Hennessey, J., "Register Allocation by Priority-Based Coloring", ACM SIGPLAN '84 Symposium on Compiler Construction, pp. 222-232, 1984.
- [8] Deo, N., *Graph Theory*, Prentice-Hall of India, New Dehli, India, 1987.
- [9] Golumbic, C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, NY., 1980.
- [10] Golumbic, C., "Interval Graphs and Related Topics", *Discrete Mathematics*, No. 55, pp. 113-121, 1985.
- [11] Gupta, R., Soffa, M.L., Steele, T., "Register Allocation Via Clique Separators", ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 264-274, 1989.

- [12] Gupta, U.I., Lee, D.T., Leung, Y.T., "Efficient Algorithms for Interval Graphs and Circular-Arc Graphs", *Networks*, Vol. 12, pp. 459-467, 1982.
- [13] Hendren, L., Gao, G., Altman, E., Mukerji, C., "A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs", *Journal of Programming Languages*, Vol. 1, No. 3, pp. 155-185, 1993.
- [14] W. -m, Hwu, W., et. al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *The Journal of Supercomputing*, Vol. 7, No. 1, pp. 229-248, 1993.
- [15] Nickerson, B., "Graph Coloring Register Allocation for Processors with Multi-Register Operands", *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 40-52, 1990.
- [16] Shimon, E., *Graph Algorithms*, Computer Science Press, Rockville, Mo., 1979.
- [17] Wolfe, M., *Analysis and Optimizations for Modern Architectures*, Oregon Graduate Institute, Beaverton, Or., 1993.

BIOGRAPHICAL INFORMATION

Lynn Thompson was born in Portland Oregon on November 29, 1959. Lynn graduated from Portland State University, in the Spring of 1983 with Bachelor of Science degree in Sociology. Studies in computer science, and mathematics continued at Portland Community College from 1984 to 1986 leading to an Associates degree in Computer Science. Employment at Intel Corporation provided incentive and means to pursue a Master's Degree in Computer Science. Lynn is employed as a Software Engineer in the Microcomputer Software Labs at Intel.