MACHINE INDEPENDENT REGISTER ALLOCATION FOR THE IMPACT-I C COMPILER

BY

RICHARD EUGENE HANK

B.S, University of Illinois at Urbana-Champaign, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Wen-mei Hwu, for his guidance and support. I have learned a great deal from him over the past few years and I look forward to working with him in the future.

I wish to extend my appreciation to Pohua Chang, Scott Mahlke, William Chen, and Roger Bringmann. Pohua's many suggestions helped a great deal during the implementation of the current register allocator. William Chen and Scott Mahlke answered my many questions and provided their sage advice during the development of both the HP PA-RISC code generator and the register allocator. Roger Bringmann's help was instrumental in the design of the machine independent interface and its integration into the IMPACT code generation template.

I also would like to thank my brother Jeff and my many friends, especially Bill, Jeremy, and Sabrina, for their friendship and moral support. They have all made my life in graduate school very enjoyable.

Finally, and most importantly, I extend my heartfelt appreciation to my parents, for without their love and guidance I would not be here today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Reduced instruction set computers (RISC) depend upon compilers for performance. Inherent in this dependence is a conflict of interest between compiler optimizations and register allocation. Compiler optimizations are done separately from register allocation for simplicity, typically ignoring the register file of the target processor and assuming the availability of an infinite number of registers. This assumption provides the optimizer with the ability to generate as many new registers as required to hold temporary values resulting from the various optimizations. Therein lies the conflict. Current RISC processor architectures contain only a limited number of registers, usually on the order of 32 to 128. Coupling this with the constraint that instruction operands must reside within these registers makes the processor's register file an important yet scarce resource. The register allocator must then attempt to make the most efficient use of the limited number of available registers so as to minimize the number of memory accesses required.

The numerous approaches to register allocation are typically divided into two classes. The first is *local* register allocation, where the scope of allocation is a basic block. The register allocator attempts to obtain an optimal allocation of temporaries found only within segments of straight line code. In the second class, *global* register allocation, the scope of allocation is enlarged to an entire function or possibly the entire program. Global register allocators, utilizing global data flow analysis techniques, attempt to obtain an optimal allocation of temporary variables, local function variables, and possibly even global variables to registers, while minimizing the number of loads and stores required.

This thesis will present the implementation of a machine independent, graph coloring register allocator within the framework of the IMPACT-I C compiler [1]. There has been some work done in the area of machine independent register allocation by Sites and Perkins [2]. Techniques for global register allocation have been extensively studied. The most notable work in the area of function level global register allocation via graph coloring has been done by Chaitin et al. [3], Chow and Hennessy [4], and more recently by Briggs [5]. A slightly different approach using hierarchical graph coloring has been proposed by Callahan and Koblenz [6]. Although this thesis will focus on intraprocedural allocation, several examples of work in the area of interprocedural or program level global register allocation are work by Chow [7], Wall [8], and Santhanam and Odnert [9].

Chapter 2 describes the register allocation problem and provides a detailed discussion of the graph coloring methods of Chaitin et al. [3] and Chow and Hennessy [4]. Chapter 3 presents the implementation details of the IMPACT register allocator. Finally, Chapter 4 contains conclusions and future work.

2. BACKGROUND

The register allocation problem involves finding an optimal assignment of function variables to the available processor registers within the constraints imposed by hardware and/or software register usage conventions. Even without these constraints, the problem is know to be NP-complete [10]. Numerous approaches have been proposed to solve this problem. This thesis will focus on the reduction of register allocation to a graph coloring problem.

The graph coloring problem is to determine the minimum number of colors needed to color a graph $G = (v, e)$, such that no two adjacent nodes have the same color. The reduction of register allocation is as follows. Each vertex v_i represents the portion of the function f where the variable i is live, i.e., its live range. An edge e_{ij} connects vertices v_i and v_j if variables i and j interfere, meaning that variables v_i and v_j are simultaneously live. If we now assume that the target processor has n registers, the register allocation problem is to obtain an n-coloring of the graph G. In the event that the graph G is not n-colorable, some node(s) must be spilled. The proposed heuristic methods for graph coloring based register allocation strive to minimize the number of spilled nodes and the resulting spill code.

The following sections describe the work done by Chaitin et al., as well as work done by Chow and Hennessy. In addition, several extensions proposed to improve the quality of the register allocation and reduce spill code will be presented.

2.1 Chaitin's Register Allocator

Register allocation via graph coloring was first implemented by Chaitin et al. in the PL.8 compiler [11], [3], [12]. Register allocation in the PL.8 compiler takes place after code generation. Delaying allocation until after code generation eliminates the need to reserve registers for usage convention reasons or as scratch registers for later stages of code generation. We will see that both Chow's register allocator and the IMPACT-I register allocator do not have this advantage. Prior to register allocation, local and temporary function variables are assumed to be located in virtual registers. This has the effect of excluding global variables as allocation candidates since spilled variables will be placed into the local stack frame. The register allocation process consists of five phases: live range construction, interference graph construction, coalescing, spill cost estimation, and coloring.

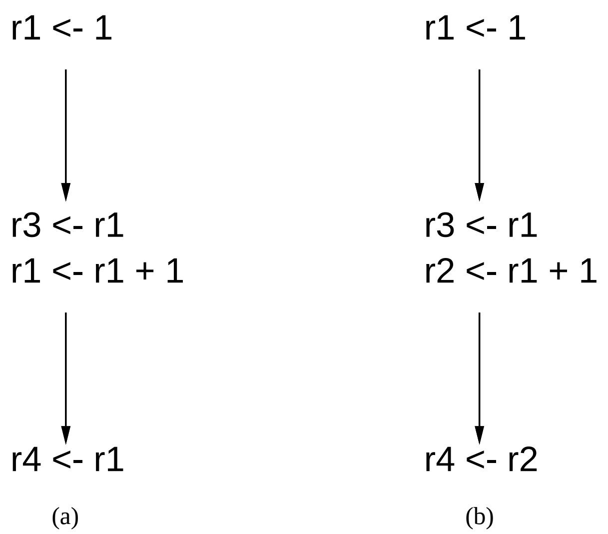

Figure 2.1: Disjoint *def-use* chains: (a) live range of r1 with two disjoint *def-use* chains, (b) result of renumbering

2.1.1 Live range construction

Before register allocation can be performed, the live range of each virtual register must be determined. Live ranges are constructed by first performing a *def-use* analysis on the function. The live range of a virtual register consists of the instructions contained within the connected components of *def-use* chains of that virtual register. A live range may, at this point, contain disjoint *def-use* chains (see **r**1 in Figure 2.1(a)). These disjoint *def-use* chains need not be allocated to the same register since they are independent. Forcing this to happen may place unnecessary constraints on the coloring process which may produce suboptimal results. Disjoint *def-use* chains within the same live range are therefore renumbered to ensure that each has a unique name. A simple example is shown in Figure 2.1(b), where the second *def-use* chain of **r**1 has been renumbered **r**2.

2.1.2 Interference graph construction and coalescing

The interference graph is the core data structure around which the entire register allocation process takes place. The interference graph consists of one node for each live range created during the previous renumbering step. The graph also contains the arcs representing interferences between two different live ranges. Simply put, two live ranges are said to interfere if one is live at the definition point of the other. Since the interference graph is the one that will be colored, an efficient implementation of this data structure is desirable. Within Chaitin's allocator, the interference graph is actually implemented as two distinct data structures: a triangular bit matrix and adjacency vectors. The triangular bit matrix allows for quick random access to the graph, whereas the adjacency vectors provide quick sequential determination of all nodes interfering with another.

Once the interference graph has been built, a process known as coalescing is performed. Coalescing involves locating noninterfering live ranges which are the source and destination of copy instructions and then merging these two live ranges into one. An example of this procedure is shown in Figure 2.2. The unnecessary copy **r**2 **<-** r1 in Figure 2.2(a), may be removed by coalescing **r**1 and **r**3, since their live ranges do not interfere. The result is shown in Figure 2.2(b). For the same reason, **r**3 and **r**6 in Figure 2.2(b) may be coalesced, resulting in Figure 2.2(c). One may notice from the figure that the result of coalescing is identical to applying copy propagation to the same code. The coalescing of nodes provides an easy method of forcing two live ranges to be allocated to the same register. Of course, once two live ranges have been coalesced, the interference

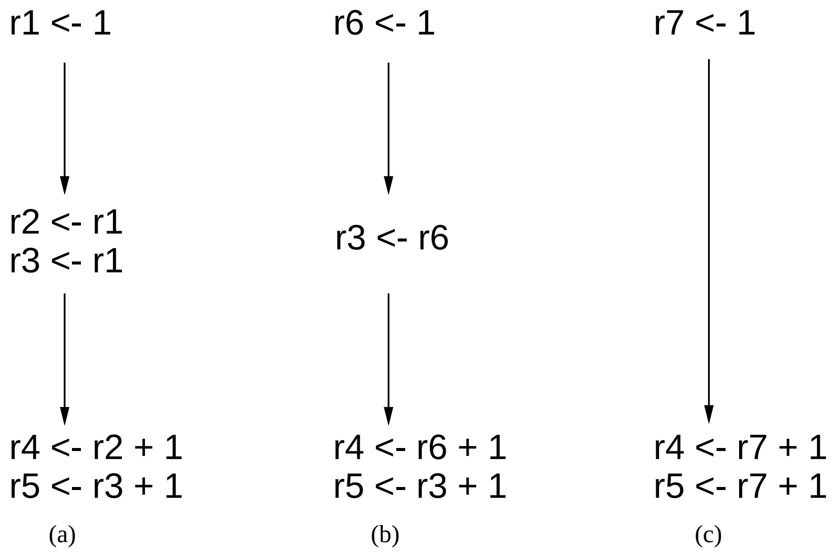

Figure 2.2: Coalescing of live ranges: (a) original live ranges, (b) coalescing of r1 and r2, (c) coalescing of r3 and r6

graph is no longer correct and must be rebuilt. The interference graph construction and coalescing steps are repeated until there are no more live ranges that may be coalesced. In practice, it has been shown that this process will converge within 2 to 3 iterations [3], [12], [5].

2.1.3 Spill costs

Now that the interference graph is stable, a spill weight is calculated for each remaining live range. Chaitin's register allocator assigns a weight to each live range that represents the cost of spilling it, which is the cost of executing the loads and stores that must be inserted if the live range were to be spilled. The execution cost of an instruction is estimated to be 10^d, where d is the loop nesting level of the instruction. The coloring process to follow will invariably have to spill a node if a register is unavailable and these estimates are used to select the live range to be spilled.

2.1.4 Coloring and spilling

The actual coloring process in Chaitin's allocator is relatively simple. If there exists a node v, such that the $deg(v) < n$, assuming a target processor with n registers, it and all of its interferences are removed from the graph and placed on a stack. Since $deg(v) < n$, the coloring of the remaining graph is independent of the color that will be assigned to this live range and it may be safely eliminated from the interference graph. If there are no nodes with $deg(v) < n$, then a node is chosen to be spilled. The heuristic used by Chaitin's allocator to select spill nodes attempts to minimize the spill instructions inserted and maximize the number of interferences removed from the graph. Thus the node with the minimum value of Eq. (2.1) is removed from the graph along with its interference arcs.

$$\frac{cost(v)}{deg(v)} \tag{2.1}$$

The resulting spill code will alter the interference graph, but rather that rebuilding the interference graph after each spill, Chaitin's allocator allows several nodes to be selected for spilling. The interference graph is then rebuilt and the coloring process starts over.

As an alternative to spilling a live range, Chaitin introduces the idea of rematerialization. That is, if a computation can be done in a single operation and its operands are always available, such as the loading of integer constants or generating frame pointer offsets, the value can be rematerialized rather than stored and reloaded from memory. There is one restriction, the use of the value being rematerialized must be reachable by definitions of identical value. If the use is reached by multiple values, the live range must be spilled.

These simplification and spilling steps continue until the graph is empty. At this point, the actual coloring is done. Nodes are popped off the stack in the opposite order in which they were removed from the graph and reinserted. Each node is assigned a color different from that already assigned to each of its neighbors within the graph. There will be a color available since a node was removed from the graph and placed on the stack if and only if a color was known to be available.

This method of graph coloring does not necessarily produce the best allocation in all situations. There have been several enhancements proposed to improve the allocation results of Chaitin-style register allocators. These are presented in the following section.

2.1.5 Spill code reduction

To illustrate a limitation of Chaitin's coloring method, consider the interference graph in Figure 2.3(a) [13]. It is obvious to the observer that a 2-coloring of this graph exists. One possible 2-coloring is shown in Figure 2.3(b). If, however, we apply Chaitin's coloring heuristic to this graph, the heuristic will discover that there exists no node with $deg(v) < 2$. Thus at least one node of this graph will be unnecessarily spilled. Figure 2.3(c) shows a possible result of applying Chaitin's allocator to this graph; **a** has been spilled and **b**,

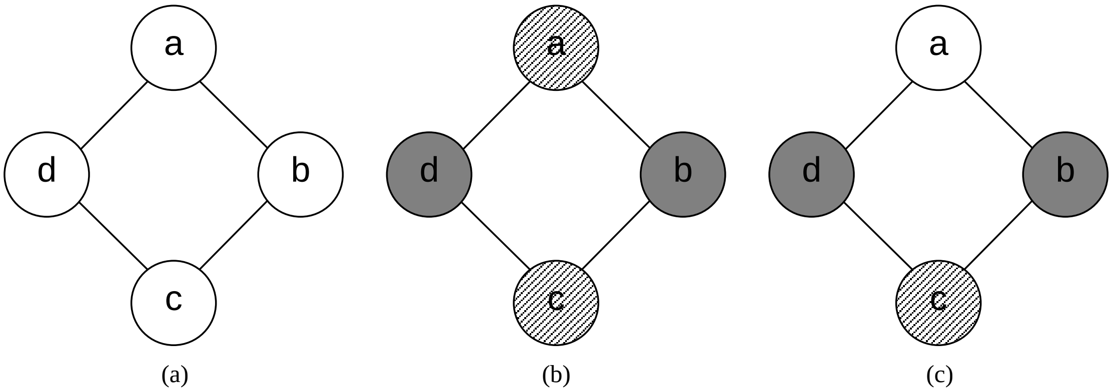

Figure 2.3: 2-colorable graph: (a) uncolored graph, (b) valid 2-coloring, (c) result of Chaitin's allocator, **a** was spilled

c, and **d** have been colored. An extension to Chaitin's method has been proposed by Briggs et al. to solve this problem [13].

Instead of spilling as the Chaitin allocator will do when there are no nodes with $deg(v) < n$, Briggs et al. select a node to place on the stack without spilling it. The coloring algorithm is now assuming that there will be a color available when the node is popped off the stack. This leads to the second extension. When a node is popped off the stack, and there is no color available, the node is left uncolored and coloring continues.

If at the end of the coloring process, there are uncolored nodes, those nodes are spilled and the necessary spill code is inserted. The interference graph is then reconstructed and the coloring process starts over.

A 2-coloring of the graph in Figure 2.3(a) can be generated using this approach. Since there is no node with $deg(v) < 2$, one node will be placed on the stack without spilling it, for instance node **a**. The remaining nodes all have $deg(v) < 2$ and are placed on the stack in some order. Once the graph is empty, the nodes are popped off the stack and assigned colors. When node **a** is popped off the stack, it will be assigned the color assigned to node **c**. The resulting 2-coloring is shown in Figure 2.3(b). Briggs' work shows that by using this method, one may obtain an allocation as good as or better than that produced by Chaitin's method.

Briggs et al. have also done work in the area of rematerialization [14]. As previously mentioned, Chaitin's allocator will correctly perform rematerialization on a live range that contains a single value. However, if the live range contains uses that are reached by multiple definitions of different values, Chaitin's allocator will simply spill the live range, if there is no available register. Briggs' work allows for the application of rematerialization to these multivalued live ranges.

A heuristic proposed by Bernstein et al. [15] seeks to reduce the spill code by improving the selection of spill nodes. When Chaitin's method chooses to spill, it selects the graph node with the minimum value of Eq. (2.1). Bernstein proposes the use of three different metrics:

$$\frac{cost(v)}{deg(v)^2}, \quad \frac{cost(v)}{area(v)deg(v)}, \quad \frac{cost(v)}{area(v)deg(v)^2} \tag{2.2}$$

where

$$area(v) = \sum_{i \in lr} 5^{depth(i)} \times width(i) \tag{2.3}$$

The depth of an instruction, $depth(i)$, is defined to be the loop nesting level of the instruction, and the width of an instruction, $width(i)$, is the number of variables live during the execution of that instruction. The idea is to run Chaitin's algorithm with each of the three new metrics and choose the best allocation of the three, that being the allocation with the least amount of spill code. The use of this heuristic has been shown to improve execution time by 2-3%.

2.2 Chow's Register Allocator

The register allocator implemented by Chow and Hennessy [16], [4], like Chaitin's allocator, utilizes graph coloring, but the similarities stop there. In Chow's scheme, register allocation is performed on an intermediate language prior to final code generation, unlike Chaitin's allocator which performs allocation after code generation. This requires that a certain number of registers be reserved as temporaries which may or may not be required by the subsequent code generation phases. Chaitin's allocator assumes that all variables reside in virtual registers. Chow's allocator, on the other hand, assumes all variables reside in some home memory location. This assumption has several ramifications for the register allocator. First, in addition to all local and temporary function variables, global variables are now candidates for allocation, whereas in Chaitin's method, global variables must be loaded into virtual registers before register allocation. Second, the code is executable even without register allocation. Third, this assumption introduces an aliasing problem, which will be discussed later. The following sections describe the Chow's register allocation algorithm.

2.2.1 Live ranges and interference

Live ranges are determined by first performing live-variable and reaching-definition analyses. From this information, the live range of a variable v is $live(v) \cap reach(v)$. This live range consists of a set of basic blocks, rather than a set of instructions as in Chaitin's allocator. Also, Chow's allocator makes no attempt to separate any disjoint *def-use* chains into separate live ranges. They chose to let their splitting mechanism achieve the same effect, though only when required. The larger granularity of the basic block representation is not seen as a limitation due to its typically small size. This small size is ensured by a compiler mechanism which breaks up large basic blocks.

Once live ranges have been constructed, the problem of aliases must be dealt with. Since every variable is assumed to have a home location, the contents of the register to which a variable is allocated may be inconsistent with its home location. If this location is indirectly accessed, incorrect values may be read. To alleviate this problem, portions of a live range where an alias may occur are removed, requiring that additional loads and stores be inserted if the live range is allocated.

Using the live-range information, an interference graph is constructed. Conceptually, there is one node for each live range and an arc is added between two nodes if they interfere. Two live ranges are said to interfere if the intersection of their live ranges is not empty. The interference graph itself is implemented by an adjacency list attached to the live-range information data structure. No coalescing is done since register allocation is done prior to code generation and previous global optimization phases perform copy propagation which achieves the same result.

2.2.2 Priorities

In Chaitin's allocator, a cost function is used to select nodes to spill when the graph contains nodes with $deg(v) \geq n$. Chow's allocator takes the opposite approach. A priority function is used to select the next node to be assigned a color. Thus, the cost/priority function used is an estimate of the execution time saved by the removal of the loads and stores already present in the code when a live range is allocated. The priority function is shown below:

$$P(lr) = \frac{S(lr)}{N} = \frac{\sum_{i \in lr} s_i \times w_i}{N} \tag{2.4}$$

where

$$s_i = (LODSAVE \times uses_i + STRSAVE \times defs_i - MOVCOST \times n_i) \tag{2.5}$$

The priority function sums the benefit $s_i \times w_i$ for each basic block i contained within the live range divided by the number of basic blocks in the live range. The value w_i is an estimate of the execution frequency of basic block i. The value s_i is the time saved by removing loads and stores minus the cost of any required moves added to basic block i. Dividing the benefit by the number of basic blocks, N, gives higher priority to shorter live ranges which tie up valuable resources for shorter periods of time. The use of this function is described in the following section.

2.2.3 Coloring, spilling, and splitting

Conceptually, the coloring algorithm in Chow's allocator is also relatively simple. The coloring algorithm deals with two kinds of live ranges: constrained and unconstrained. A live range is unconstrained if it has degree less than the available number of registers and constrained otherwise. The first step of the process is to remove the unconstrained live ranges from the interference graph. The priority function, Eq. (2.4), is then computed for each of the remaining constrained live ranges. If the priority function is negative or if the live range is uncolorable, meaning all available registers have been assigned, that live range is removed from the graph. The live range of highest priority is selected and assigned a color. If there is no available color, an attempt is made to split the live range. The processes of calculating needed priority functions and selecting the node of highest priority are repeated until the graph is empty. Note that this coloring algorithm, as presented, will obtain a 2-coloring of the interference graph in Figure 2.3.

When there is no available register for the selected live range, the register allocator attempts to split it. The basic idea behind live-range splitting in Chow's allocator is to find the largest allocatable portion of an unallocatable live range. The first task in generating the new live range is to locate a starting point. The algorithm selects a basic block where the live range begins with a definition; if this is not possible the first basic block containing a use is selected. Next, a breadth-first traversal of the original live range is performed and basic blocks are added to the new live range until it is no longer allocatable. Once the two live ranges have been determined, the interference graph must

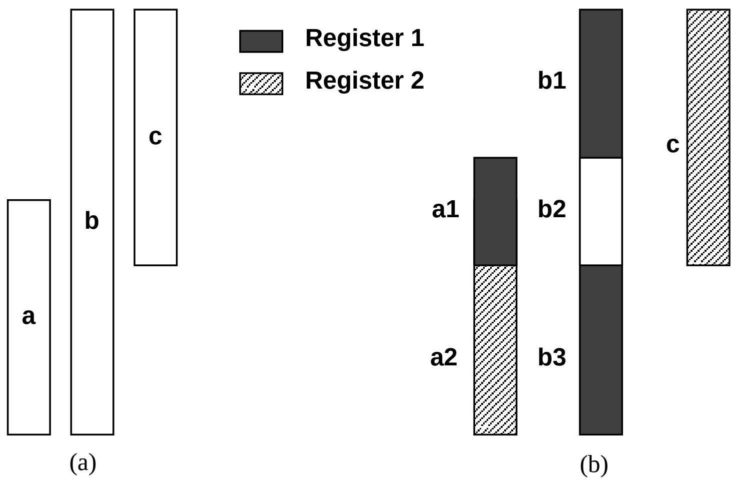

Figure 2.4: Live range splitting example: (a) live ranges a, b, and c prior to allocation, (b) live ranges a, b, and c after live range splitting, b2 is spilled

be updated to correct the interference lists of those live ranges that interfered with the original live range. In addition, spill code must be inserted along the split boundary to ensure correctness. Finally, since the two new live ranges and/or unconstrained live ranges interfering with them may become constrained, the unconstrained and constrained live-range sets must be updated. Figure 2.4(a) contains a set of three live ranges. If the target processor has only two registers, Figure 2.4(b) shows one possible result of Chow's splitting algorithm. Live range **a** has been split between registers 1 and 2, while most of **b** has been allocated to register 2, but a portion of it, **b**2, was spilled. The are numerous other possibilities depending upon the relative priorities of each of the live ranges.

2.2.4 Extensions

The above coloring algorithm may also be applied to registers with different saving conventions, i.e., caller-saved and callee-saved registers. This is accomplished by providing two functions for each live range: a priority value if allocated to a caller-saved register and a priority value if allocated to a callee-saved register. The priority value used in the above algorithm is then the maximum of the two. Further work has been done by Chow to extend this method to interprocedural register allocation [7]. The idea is to minimize the amount of caller-saved code at procedure calls and callee-saved code upon entry, by making available to the register allocator the register usage of all functions called by the function being allocated. This is accomplished by allocating the functions of a program in a bottom-up traversal of the program's call graph. The chief drawback to this approach is that all functions must be made available to the compiler at compilation time.

A Chow-style register allocator has also been implemented in the SPUR Lisp Compiler by Larus and Hilfinger [17]. The SPUR allocator differs in that all temporaries are assumed to be in virtual registers similar to a Chaitin-style allocator and spill code must be inserted if such a live range is spilled. Also, register allocation takes place after code generation on SPUR assembly code. Neither of these differences is particularly significant from a register allocation standpoint.

3. REGISTER ALLOCATION IN THE IMPACT-I C COMPILER

3.1 Allocation Environment

The compilation framework of the IMPACT-I C compiler consists of numerous independent programs, each of which performs a distinct function in the transformation of a C program into the assembly language of one of several target processor architectures. The final stage of the compilation process is, of course, code generation for the target processor. Currently, code generators have been built for the Am29000^{TM}, HP PA-RISC^{TM}, Sun Sparc^{TM}, and MIPS R2000^{TM} processors. It is during this stage of compilation that register allocation is performed.

A detailed description of the IMPACT code generation template may be found in R. A. Bringmann's thesis [18]. A brief description is provided here to present an overview of the environment in which register allocation takes place and to illustrate one of the primary requirements the current register allocator had to fulfill, that being machine independence. The intermediate language of the IMPACT-I C compiler is Lcode, which is a machine independent, RISC-like assembly code similar to that of the MIPS R2000. The task of the code generator is to translate Lcode to the assembly language of the target processor. Code generation is performed in three phases. In phase 1, Lcode is translated to Mcode (or machine specific Lcode). The goal here is to annotate Lcode into a one-to-one mapping with the final assembly code using the architectural constraints of the target processor. Phase 2 involves a pass of local optimizations: common subexpression elimination, copy propagation, and dead code removal; prepass code scheduling; register allocation; machine specific peephole optimizations; and postpass code scheduling. Finally, phase 3 takes as input the results of phase 2 and emits assembly code of the target processor.

The register allocator interface allows the same allocator to be utilized by all of the code generators. This makes the job of the code generator writer easier and at the same time allows us to evaluate the performance of the register allocator across a wide variety of processor architectures. The register allocator takes as input Lcode/Mcode, assuming all local variables and temporaries are located in virtual registers, as in Chaitin's allocator, and returns an allocated Lcode/Mcode function including any required spill code.

The following sections describe the register allocation interface and present the implementation details of the current IMPACT register allocator.

3.1.1 Internal register file representation

The IMPACT register allocator like the aforementioned allocators attempts to obtain an n-coloring of some arbitrary interference graph. For Chaitin's allocator, n represents 16 [3] or 32 [12] general purpose processor registers, and for Chow's allocator, n represents the register file of the MIPS R2000 processor [4]. For our purposes, n is an array of an arbitrary number of typeless, unit-sized general purpose registers. Allocation of registers from this array allows the register allocator to be completely independent of the processor for which allocation is being done. By partitioning this array into various configurations, the register allocator may effectively handle instances of type overlap, where for example integer and floating-point data may reside within the same registers as in the Am29000, as well as handle different register saving conventions, such as caller-saved and callee-saved registers. Allocation for virtually any RISC processor architecture may be accomplished by simply mapping the register file of the target processor to the internal array used by the allocator.

This mapping is achieved by partitioning the target processor's register file into a number of "register banks." Each register bank is uniquely defined by two attributes. The first and most obvious is the register data type which may be, but is not limited to, one of those shown in Table 3.1. The second is the register saving convention as in Table 3.2.

Since each bank is uniquely defined by these two attributes, there can be at most one bank for each combination. Current RISC implementations have at most one bank

Table 3.1: Register allocator internal data types

Internal Type	**D**ata Type
R_INT	integer
R_FLOAT	single-precision floating-point
R_DOUBLE	double-precision floating-point
R_PREDICATE	predicate

Table 3.2: Register allocator saving conventions

Internal Type	**S**aving Convention
R_CALLER	caller-saved registers
R_CALLEE	callee-saved registers

of registers for each data type so this constraint is not as limiting as it may seem. The implementation details for multiple banks of the same attribute, for example, two integer caller-saved banks, involve changes to instruction scheduling as well as register allocation and are beyond the scope of the current implementations of both the scheduler and the register allocator.

3.1.2 Machine-independent interface

The register file of the Hewlett-Packard PA-RISC 1.1 [19], [20] processor will serve as an example to better illustrate how to map a processor register file to the register allocator array. The PA-RISC 1.1 register file is shown in Table 3.3. The register file will be divided into six banks, one for each available data type and calling convention combination. For example, let **B**ank 1 consist of the integer caller-saved registers; there are four of them (gr1,gr19-gr22 in Table 3.3). The next bank, **B**ank 2, will comprise the integer callee-saved registers; there are 16 of them (gr3-gr18 in Table 3.3). The complete

Table 3.3: Register file of the PA-RISC 1.1

PA-RISC 1.1 Register File	
Register	**U**sage
gr0	Zero value register
gr3-gr18	Callee saved registers
gr1,gr19-gr22	Caller saved registers
gr23-gr26	Parameter registers
gr2,gr27-gr31	$sp, $dp, return value, etc.
fr0-fr3	Coprocessor status, exception registers
fr4-fr7	Parameter registers
fr8-fr11	Caller saved registers
fr12-fr21	Callee saved registers
fr22-fr31	Caller saved registers

set of bank definitions is shown in Table 3.4. The contents of the **C**lass, **T**ype, and **N**um rows are easily determinable from Table 3.3. The **S**ize attributes are architecture dependent and for the PA-RISC the integer and floating-point banks have size 1, since an integer or floating-point register requires one element of the array, but the double banks, on the other hand, are given size 2. This is due to the fact that a double-precision register in the PA-RISC 1.1 may hold one double-precision value or two single-precision values; thus, each double-precision register requires two elements of the array. The value provided for the **O**ffset attribute serves only to separate disjoint register banks or merge overlapping banks. The mapping created by the offset values in Table 3.4 is given in Figure 3.1. Note that the offsets given to **B**ank 1 and **B**ank 3 serve to separate them in the mapping in Figure 3.1, and the offsets of **B**ank 2 and **B**ank 3 are identical, because they are in reality the same registers.

Table 3.4: Register bank partition for PA-RISC 1.1

Attribute	**B**ank 1	**B**ank 2	**B**ank 3
Class	R_CALLER	R_CALLER	R_CALLER
Type	R_INT	R_FLOAT	R_DOUBLE
Offset	0	20	20
Num	4	28	14
Size	1	1	2
Attribute	**B**ank 4	**B**ank 5	**B**ank 6
Class	R_CALLEE	R_CALLEE	R_CALLEE
Type	R_INT	R_FLOAT	R_DOUBLE
Offset	4	48	48
Num	16	20	10
Size	1	1	2

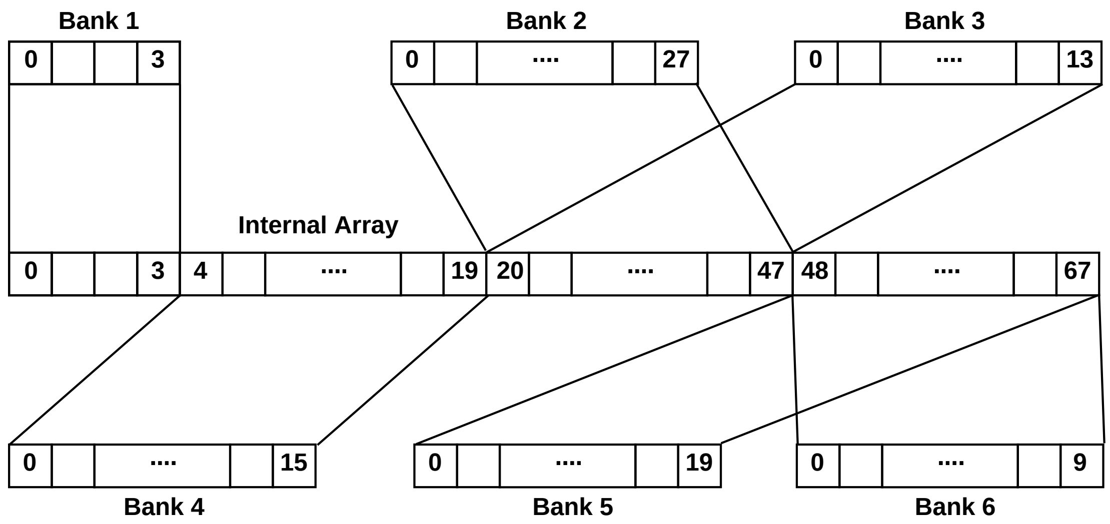

Figure 3.1: Mapping of the PA-RISC 1.1 register file

R_define_physical_bank(*int class, int type, int offset, int num, int size, int *map*)

Figure 3.2: Processor register bank definition function

```
R_define_physical_bank(R_CALLER, R_INT,    0,  4,  1, caller_int_map);
R_define_physical_bank(R_CALLEE, R_INT,    4,  16, 1, callee_int_map);
R_define_physical_bank(R_CALLER, R_FLOAT,  20, 28, 1, caller_double_map);
R_define_physical_bank(R_CALLEE, R_FLOAT,  48, 20, 1, callee_double_map);
R_define_physical_bank(R_CALLER, R_DOUBLE, 20, 14, 2, caller_double_map);
R_define_physical_bank(R_CALLEE, R_DOUBLE, 48, 10, 2, callee_double_map);
```

Figure 3.3: Register allocator interface for PA-RISC 1.1

The information for each register bank is provided to the register allocator through the function **R**_define_physical_bank whose prototype is shown in Figure 3.2. The parameters correspond exactly to each of the five attributes in Table 3.4 except for the last one which is an array of names. This "map" allows the register allocator to associate a name with each register in a register bank. These are the names placed in the Lcode/Mcode function once allocation is complete. The complete series of function calls required to describe the register file of the PA-RISC 1.1 processor are provided in Figure 3.3. The register maps referenced by this series of function calls are provided in Appendix A.

The register file of the Advanced Micro Devices Am29000 [21] (provided in Table 3.5) is described to the register allocator in essentially the same way. There are a few differences due to the fact that the Am29000 uses variable-sized register windows, and that the Am29000 register file is truly general, that is, a register may contain an integer, single-precision floating point, or half of a double-precision floating-point value. The differences are apparent in the bank definitions in Table 3.6. The offsets are identical for each of

Table 3.5: Register file of the Am29000

Am29000 Register File	
Register	**U**sage
gr1	Reserved
gr2-gr63	Unimplemented
gr64-gr95	Reserved for OS
gr96-gr97	Return value, pointer
gr98-gr111 gr116-gr120	Caller-saved Registers
gr112-gr115 gr121-gr127	Reserved
1r0-1r1	Return address, frame pointer
fr2-fr128	Callee-saved registers (# varies)

the three caller-saved banks and for each of the three callee-saved banks, indicating that integer, single-precision, and double-precision values may reside in the same registers. Also, note that the **N**um attributes for the callee-saved banks are variable; these are the registers in the variable-sized register window. Each function has available to it 126 registers minus the number of incoming (I) and outgoing (O) parameters. The resultant mapping is shown in Figure 3.4, the required function calls are shown in Figure 3.5, and the register maps referenced by these calls are provided in Appendix A. Since the number of registers available in the window varies for each function with the number of incoming and outgoing parameters, the last three function calls in Figure 3.5 must be repeated prior to allocation of each function to update the register allocator's register file description.

As implemented, the internal array used by the allocator is in effect a virtual array, it does not exist. Instead, there is an array of structures which describe the mapping of the target processor register file onto the virtual array. This structure is shown in

Table 3.6: Register bank partition for Am29000

Attribute	**B**ank 1	**B**ank 2	**B**ank 3
Class	R_CALLER	R_CALLER	R_CALLER
Type	R_INT	R_FLOAT	R_DOUBLE
Offset	0	0	0
Num	19	19	9
Size	1	1	2
Attribute	**B**ank 4	**B**ank 5	**B**ank 6
Class	R_CALLEE	R_CALLEE	R_CALLEE
Type	R_INT	R_FLOAT	R_DOUBLE
Offset	19	19	19
Num	126-I-O	126-I-O	(126-I-O)/2
Size	1	1	2

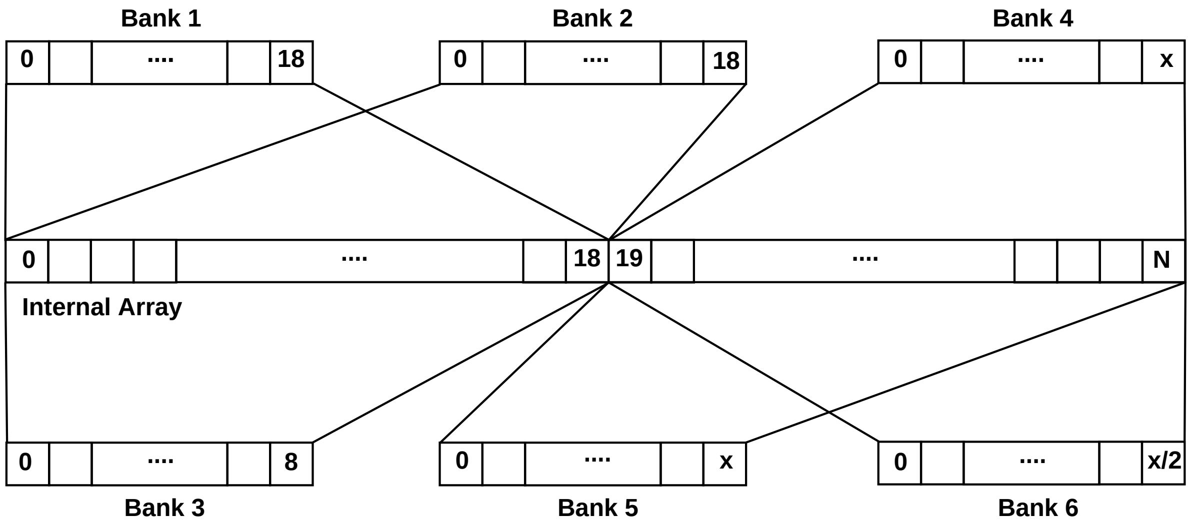

Figure 3.4: Mapping of the Am29000 register file

```
R_define_physical_bank(R_CALLER, R_INT,     0,  19,  1, caller_reg_map);
R_define_physical_bank(R_CALLEE, R_INT,     0,  19,  1, callee_reg_map);
R_define_physical_bank(R_CALLER, R_FLOAT,   0,  9,   1, caller_reg_map);
R_define_physical_bank(R_CALLEE, R_FLOAT,   19, x,   1, callee_reg_map);
R_define_physical_bank(R_CALLER, R_DOUBLE, 19, x,   2, caller_reg_map);
R_define_physical_bank(R_CALLEE, R_DOUBLE, 19, x/2, 2, callee_reg_map);
```

Figure 3.5: Register allocator interface for Am29000

```
typedef struct R_Physical_Bank {
    short defined;
    short class;
    short type;
    short base_index;
    short num_reg;
    short reg_size;
} R_Physical_Bank;
```

Figure 3.6: Register bank definition structure

Figure 3.6 and the fields correspond to the five previously defined attributes. Each call to **R**_define_physical_bank initializes one of these structures. Exactly how these are used during the graph coloring process will be described later.

3.2 Live-Range Determination

Live ranges are constructed by performing live variable analysis and reaching definition analysis like Chow's algorithm. The live range of a variable v is then $live(v) \cap reach(v)$, but the live range is represented as a set of instructions as in Chaitin's allocator. The actual algorithm for live-range determination is given in Figure 3.7. The larger granularity of a basic block representation is insufficient when allocation is being performed on superblocks [22] or hyperblocks [23], where the typical block size is not small. There is no attempt to renumber disjoint *def-use* chains as our work has shown the benefits to be negligible. These results will be presented in Section 3.11. The algorithms used to perform dataflow analysis are the basic algorithms found in Aho, Sethi and Ullman [24]. Although not necessarily the most time efficient method for dataflow analysis, these algorithms are exact and are easily implemented. Improving our methods

```
for  ( each instruction i ) {
      live_var = i->def ∪ i->in
      for ( each v ∈ live_var )
            Set_add(live(v),i)
      reach_def = instr->reach_def
      for ( each instr j ∈ reach_def )
            Set_add(reach(j->def),i)
}
for ( each virtual register v )
      live_range(v) = live(v) ∩ reach(v)
```

Figure 3.7: Live-range determination

of dataflow analysis is slated for the future, but this will simply serve to speed up the allocation, not improve the allocation results.

3.3 Interference Graph

The interference graph consists of a group of nodes linked together as both a red/black tree and a linked list. The red/black tree provides random access in a reasonable amount of time, $O(log_2 n)$ in the size of the graph, while the linked list provides fast sequential access. Each graph node consists of the name of the virtual register it represents, the priority value of its live range, which will be discussed later, and an adjacency list of pointers to graph nodes of interfering live ranges.

The interferences themselves are generated by comparing pairs of virtual registers. If the intersection of their live ranges is not empty, an interference arc is added to the adjacency list of each graph node. The algorithm for constructing the interference graph is given in Figure 3.8 and works as follows. A graph node is created for each live range

```
for ( each live range ) {
      node = NewNode()
      < initialize graph node >
      AddNode(i_graph,node)
}
for ( n1 = each graph node )
      for ( n2 = each graph node after n1 )
            if ( live_range(n1) ∪ live_range(n2) ≠ ∅ ) {
                  Connect_Nodes(n1,n2)
                  Connect_Nodes(n2,n1)
            }
```

Figure 3.8: Interference graph construction

and inserted into the graph by **A**ddNode which inserts the new node into the linked list and the red/black tree. Then, each pair of live ranges is compared and if they interfere, graph arcs are added to the graph nodes via **C**onnect_Nodes.

The interference graph as implemented is currently not as memory efficient as it could be, but it is extremely flexible. With this implementation, it is very easy to expand the graph when new live ranges are created; there is no need to worry about overflowing a bit matrix.

3.4 Class Preferences

Procedure call penalties can be minimized by the use of caller-saved and callee-saved registers. Typically, the register set of the processor is divided by software convention into these two classes of registers. This method may reduce procedure call penalty, but it makes the register allocator's task more difficult. Now the register allocator can not simply assign a register to a live range, it must assign a register of the appropriate convention.

The register allocator determines the preferred convention by examining the live range of each virtual register for procedure calls. Live ranges containing few or no procedure calls should be placed in caller-saved registers since little or no spill code will be required to save and restore the value around the call. Live ranges with a large number of procedure calls should be placed in callee-saved registers; thus spill code is only required if the callee uses that register. Two benefits are calculated for each live range. Each benefit represents the amount of execution time saved by allocating to a caller-saved or callee-saved register over simply spilling the virtual register. The cost of spilling a virtual register is the time required to execute the required spill code at each definition and use point within the live range. The spill cost function is

$$spill\_cost(lr) = \sum_{i \in lr} (def_i + use_i) \times w_i \tag{3.1}$$

where $def_i = 1$ iff instruction i defines the variable and $use_i = 1$ iff instruction i references the variable; otherwise, it is 0. The value w_i is the estimated execution frequency of instruction i, which may be determined statically as in Chaitin's allocator or from available dynamic profile information.

The benefit of allocating a virtual register to a caller-saved register is the *spill_cost* minus the cost of inserting spill code around any procedure calls contained in the live range.

$$caller\_benefit(lr) = spill\_cost(lr) - jsr\_weight \times caller\_cost \tag{3.2}$$

O_callee_cost(*int leaf, int callee_allocated*)
O_caller_cost(*int leaf*)

Figure 3.9: Caller/callee cost interface functions

The jsr_weight is the sum of the execution frequencies of any procedure calls in the live range. The *caller_cost* is a processor-dependent value that determines the cost of saving and restoring a value around a procedure call. This information is provided to the register allocator via the function **O**_caller_cost, which is provided by the code generator. The prototype for this function is shown in Figure 3.9.

The benefit of allocating a virtual register to a callee-saved register is the *spill_cost* minus the cost of inserting spill code at the entry and exit points of the function.

$$callee\_benefit(lr) = spill\_cost(lr) - (fn\_weight + 1) \times callee\_cost \tag{3.3}$$

The fn_weight is the estimated execution frequency of the function being allocated, which is obtained from profile information. The *callee_cost* is also a processor-dependent value that determines the cost of saving and restoring a callee-saved register. This information is provided to the register allocator via the function **O**_callee_cost, which is also provided by the code generator. The prototype for this function may be found in Figure 3.9.

Table 3.7 shows the values returned by the interface functions for all possible combinations of their parameters for the PA-RISC 1.1 and the Am29000. The first 2 columns represent the parameters of the interface functions. The parameter *leaf* is 1 if the current function contains no subroutine calls and is 0 otherwise. The parameter *callee_allocated* is

Table 3.7: Example caller/callee costs

		PA-RISC 1.1		**A**m29000	
leaf	callee_allocated	**c**aller	**c**allee	**c**aller	**c**allee
0	0	2	2	2	0
0	1	2	2	2	0
1	0	0	2	0	4
1	1	0	2	0	0

1 if a virtual register has been allocated to a callee-saved register and is 0 until that point. These costs, the remaining 4 columns, may be interpreted as a number of instructions. For example, for both processors, the caller cost of a nonleaf function is 2. This is rather intuitive since a store and a load are required to save and restore a value around each subroutine call. The callee-cost column for the Am29000 is a bit more interesting. A penalty of four instructions is incurred for the use of any number of callee-saved registers in a leaf function. This is the cost of setting up the register window, which is not required for a leaf function, whereas in the other three cases, the window is required and the use of a callee-saved register is essentially free. The one cost not requested by the allocator is the cost of using a callee-saved register that has already been used, which is, of course, 0.

3.5 Priorities

During the graph coloring process, a priority function is used to determine the virtual register that is to be assigned next. An extension to Chow's allocator allows the use of registers having different saving conventions. The priority value used to select the next virtual register to allocate is the maximum of the caller and callee priorities; see Section 2.2.4. The function used by the IMPACT register allocator, shown below, was found, by experimentation, to be a very effective metric.

$$P(lr) = \frac{spill\_cost(lr)^3}{N} \tag{3.4}$$

The *spill_cost* above is the value generated by Eq. 3.1 and N is the number of instructions within the live range of a virtual register. Dividing the function by N, serves the same purpose it does in Chow's allocator, to give priority to smaller live ranges. The priority function is actually calculated for each virtual register during the interference graph construction phase.

3.6 Coloring

The coloring process begins by removing all unconstrained virtual registers from the interference graph, i.e., those virtual registers having a number of interferences that are less than the number of available registers of that data type. The remaining virtual registers are sorted by their priority function using a heap. The use of a heap to maintain the sorted list of virtual registers has several advantages. First, heap sorts are very efficient. Second and most important, any new virtual registers generated may be added to the sorted list by simply inserting them into the heap. Maintaining the sorted list in the array would require much more time to resort the list when new virtual registers are added.

Coloring proceeds by repeatedly selecting the virtual register of highest priority, the one at the top of the heap, and attempting to assign a free register to each. As the virtual register is removed from the heap, it is placed into a list of visited virtual registers. The reason for this will be clear shortly. The internal register file is then scanned for a free register to assign to the current virtual register. The free register selection process is described in more detail in Section 3.6.1. If a free register is available, that register is assigned and a new virtual register is selected. When no free register is available, the virtual register will be spilled. The first spilled virtual register will cause the register allocator to examine the number of registers provided for spilling. If none are provided by the code generator, they are acquired as described in Section 3.6.2 and the coloring process starts over. The virtual registers in the visited list are reinserted into the heap and the coloring process starts over.

3.6.1 Free register selection

The register allocator may have a number of register banks to chose from when selecting a register for the current virtual register. The choice comes from the availability of caller-saved and/or callee-saved register banks and from the availability of macroregisters, which will be discussed in Section 3.10. There are also those registers the allocator may not use when selecting a register for the current virtual register. Those registers are any that may be reserved for other purposes, i.e., spilling, and those already allocated to interfering live ranges. The registers that are unavailable, termed reserved registers, are determined by examining each neighbor of the current live range in the graph and placing in a set those registers that have already been used. This set is then unioned with the set of registers reserved for spilling, if any.

Given the set of reserved registers, the register allocator must now determine if there is a register available for the current virtual register. The register allocator selects a register from both the caller-saved, $\mathbf{r}_r$, and the callee-saved, $\mathbf{r}_e$, register banks. The algorithm to select a free register from a particular register bank with a given set of reserved registers is shown in Figure 3.10. The selected registers are then examined and the register allocator chooses one to assign to the virtual register.

If both registers are -1, there is no choice and the register allocator spills the virtual register. If only one of the registers is -1 and the class benefit of the nonnegative register is positive, that register is assigned to the live range; otherwise, the virtual register is spilled. For example, if $\mathbf{r}_r$ = -1 and $\mathbf{r}_e > 0$, $\mathbf{r}_e$ will be assigned to the virtual register if the callee benefit of the virtual register is greater than zero. When both $\mathbf{r}_r$ and $\mathbf{r}_e$ contain valid registers, the choice is made based on the previously computed preferences. If a callee-saved register is preferred, then $\mathbf{r}_e$ is assigned to the virtual register. Also, if a caller-register is preferred but the selected callee-saved register has been used before making the callee-benefit greater than the caller-benefit, $\mathbf{r}_e$ is assigned. This may happen because once a callee-saved register has been used, it may be reused without penalty thereby raising the callee-saved benefit. Otherwise, the caller-saved register $\mathbf{r}_r$ is assigned.

```
for ( r = 0; r < reg_bank->num_reg; r++ ) {
    base = r × reg_bank->reg_size + reg_bank->base_index;
    free = 1;
    for ( s = 0; s < reg_bank->reg_size; s++ )
        if (Set_in(reserved_registers,s)) {
            free = 0;
            break;
        }
    if ( free )
        return(base);
}
return(-1);
```

Figure 3.10: Free register determination

3.6.2 Spilling

Eventually, a virtual register will be spilled and when this occurs the register allocator has to reserve a register for saving and restoring the values of this virtual register, i.e., a spill register. The actual user of the register allocator has two options. The first is to specify the registers that the register allocator is to use for spilling. This is done by providing a new register bank class, **R**_SPILL. Using this class it is possible to specify the processor registers the allocator is to use for the spilling of each data type. The number of registers provided for this purpose is completely arbitrary; if it is insufficient, the user will be informed.

The second option is to allow the register allocator to appropriate registers for this purpose. The first time a virtual register is spilled, the register allocator will examine the available spill register banks, i.e., banks with class **R**_SPILL. If the needed register banks are not defined, they are generated from the available caller-saved register banks. Under the current implementation, the register allocator will remove three registers from each caller-saved bank to form spill register banks. As an example, recall the PA-RISC 1.1 register file mapping in Figure 3.1. If a function requires the spilling of both integer and double-precision data, three registers will be removed from both the integer, caller-saved bank (**B**ANK 1) and the double-precision, caller-saved bank (**B**ank 3). A side effect of this is that six registers are moved from the single-precision, caller-saved bank (**B**ank 2). The new mapping is shown in Figure 3.11. If a number of caller-saved banks overlap, the allocator will use the same registers in the overlapping banks to minimize the number of registers lost to spilling. Thus, if single-precision spill registers were required above, they would have been selected from the same registers chosen for spilling double-precision data.

It is worth mentioning that the reservation of spill registers in this manner may produce a suboptimal allocation in some cases. This method will not affect the quality of allocation for functions where no spilling is done, and the effect on functions with extremely high register pressure will also be minimal. However, for functions that would require only a moderate number of live ranges to be spilled, the appropriation of three registers may induce more spill code than necessary. The solution is to eliminate the necessity of reserving spill registers and deal with spilling as is done in Chaitin's allocator. The benefit will be a better allocation at the cost of longer register allocation time, since coloring would be done more than once.

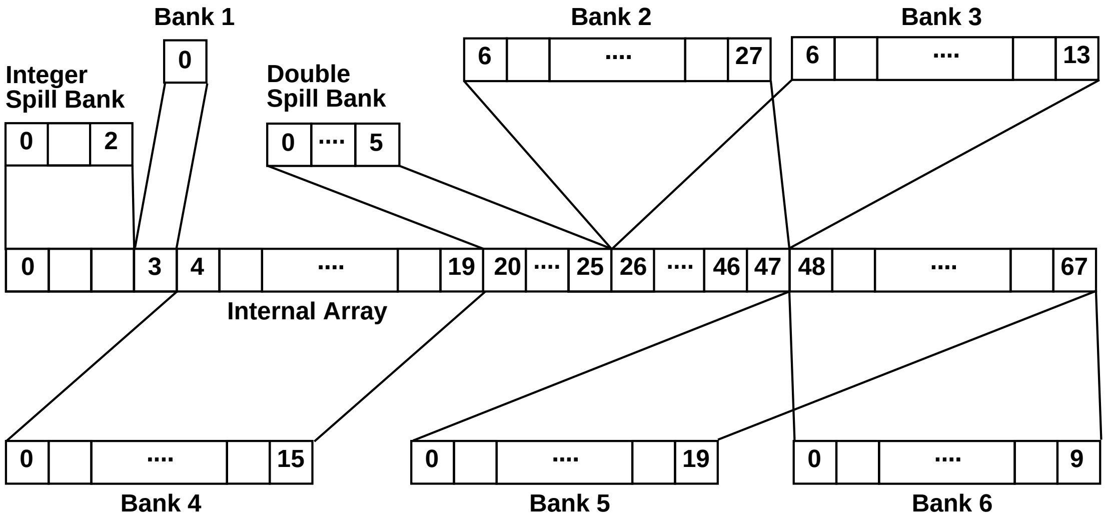

Figure 3.11: Example automatic spill register selection mapping

3.7 Machine Register Assignment

The registers assigned to virtual registers during the allocation process are simply indices into the internal array of the allocator. These numbers have no meaning to the code generator so they must be somehow mapped to the actual register file of the target processor. During the definition of each register bank, the register allocator was provided an array through the *map* parameter of the **R**_define_physical_bank function (see Figure 3.2). This array or map contains the processor registers corresponding to a particular bank. The allocator uses the index assigned to a virtual register to determine the proper index into the register and then replaces every occurrence of that virtual register with the contents of the register map location.

3.8 Spill Code Insertion

Once allocation and register assignment are complete, the register allocator may have to insert a number of load and store instructions into the function for correctness. These instructions are needed in two cases: a virtual register was spilled or a virtual register was allocated to a caller-saved register. In the case of a spilled virtual register, a store is needed after each definition and a load is needed prior to each reference within its live range, though some of these loads can be easily optimized out. When a virtual register is allocated to a caller-saved register and store is required before and a load is required after each subroutine call within the virtual register's live range.

The selection of the spill register to be used in a particular spill code instruction can greatly affect any code scheduling done subsequent to register allocation. If the same spill register is used repeatedly, the code motion available to the inserted instructions will be minimal. This can adversely affect superscalar/VLIW machines with high issue rate, 4 or 8. Obviously, post-pass scheduling will perform better if a large number of spill registers are available to the register allocator. However, reserving too many registers for spilling may induce even more spill code and further reduce the performance of the resulting code. This problem illustrates one reason to integrate register allocation and scheduling, which is not easy to accomplish [25].

Since register allocation takes place during phase 2 of code generation, the register allocator cannot simply insert the required load and store instructions. The code has been placed in an explicit format by phase 1 of code generation and the insertion of

O_spill_reg(*int reg, int type, int offset*)
O_fill_reg(*int reg, int type, int offset*)
O_mov_reg(*int dest, int src, int type*)
O_gen_branch(*int opcode, int dest*)

Figure 3.12: Instruction insertion interface functions

incorrect code may adversely affect later phases. This problem is eliminated through the same method as the caller/callee costs, via interface functions provided by the code generator. The register allocator obtains properly formatted spill code with the functions shown in Figure 3.12.

The function **O**_spill_reg returns the sequence of instructions required to spill register *reg* of type *type* to location *offset*. The opposite holds true for **O**_fill_reg. The remaining function calls are not needed unless live-range splitting is performed; they are provided here for completeness.

3.9 Base Register Allocator Evaluation

Evaluation of the quality of register allocation within the IMPACT C compiler framework is not trivial to do without another global register allocator for comparison. For this reason, the numbers presented in this section will chiefly be used as a baseline to evaluate successive techniques applied to improve the quality of the allocation achieved with the previously described coloring algorithm. The test suite used to produce these base numbers is a collection of 50 of the most frequently executed functions from a series of 12 integer and floating-point benchmark programs. Ten of the functions are from five integer programs: cccp (2 functions, eqntott (1 function), espresso (3 functions), lex (1 function), and li (3 functions). The other 40 functions are from seven members of the SPEC benchmark suite: doduc (14 functions), ear (1 function), fpppp (3 functions), matrix300 (2 functions), nasa7 (8 functions), spice (11 functions), and tomcatv (1 function).

The target architecture for these experiments is the Hewlett Packard PA-RISC 1.1 processor. The test suite was compiled to Lcode by the IMPACT C compiler at optimization level 4. The resulting Lcode is then passed to the PA-RISC code generator, which performs Lcode to Mcode annotation during phase 1 followed by machine dependent optimizations, prepass scheduling, and register allocation during phase 2. During the register allocation process, all weights and priority functions are calculated using profile information provided by the IMPACT C compiler.

The base register allocation results are shown in Table 3.8. The first two columns of this table contain the benchmark and function names. The **L**oad column contains the static number of load instructions inserted by the register allocator for spilling. The column labeled **%** immediately to the right of the **L**oad column contains the percentage of all dynamically executed instructions within the function that are loads inserted as spill code. The next two columns, **S**tore and **%**, provide the same information for stores inserted by the allocator. The **C**aller % column provides the dynamic percentage of the spill code inserted for saving and restoring registers around subroutine calls. Finally, the **T**otal % column provides the total percentage of all dynamically executed instructions that are spill code. For example, for the function **d**o_define, there were 23 loads and 87

Table 3.8: Base register allocation results

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**C**aller %	**T**otal %
cccp	do_define	23	1.07	87	4.91	0.23	5.99
	rescan	53	0.51	101	0.76	0.00	1.27
doduc	debflu	149	1.00	239	0.84	0.00	1.84
	debico	61	10.14	61	1.09	0.00	11.23
	deseco	495	5.98	497	2.09	0.00	8.07
	drepvi	14	1.70	27	3.28	0.00	4.98
	ihbtr	17	0.65	31	2.60	0.00	3.26
	lissag	5	0.01	12	0.01	0.00	0.02
	orgpar	110	0.04	100	1.94	0.00	1.98
	paroi	101	3.92	126	0.54	0.00	4.46
	pastem	51	0.01	104	0.06	0.01	0.07
	prophy	411	15.72	285	9.36	0.00	25.08
	repvid	22	5.04	17	1.09	0.00	6.13
	sortie	4	0.00	16	4.70	0.00	4.71
	subb	135	14.33	84	8.92	0.00	23.25
	supp	154	13.03	102	8.63	0.00	21.66
ear	EARSTEP	17	0.17	25	0.17	0.34	0.34
eqntott	read_ones	6	0.01	28	0.04	0.00	0.05
espresso	scofactor	10	0.41	28	2.04	0.00	2.45
	sccc	29	0.61	68	1.52	0.00	2.13
	tautology	14	0.39	35	1.24	0.00	1.63
fpppp	fmtgen	13	1.12	28	3.53	1.97	4.65
	fpppp	2495	33.98	974	13.27	0.00	47.25
	twldrv	805	12.86	466	5.23	0.00	18.08
lex	packtrans	11	0.02	42	0.13	0.00	0.15
li	evform	49	0.97	60	2.83	0.48	3.80
	xlevlist	22	0.96	27	2.07	0.92	3.02
	xlevarg	21	0.84	27	2.85	0.00	3.69
matrix300	sgemm	16	0.08	5	0.07	0.00	0.15
	sgemv	14	0.04	11	0.00	0.00	0.04
nasa7	cfft2d1	32	0.03	24	0.01	0.00	0.04
	cfft2d2	28	0.05	18	0.01	0.00	0.06
	cholsky	80	0.17	41	0.02	0.00	0.19
	btrix	140	7.34	91	1.12	0.00	8.46
	gmtry	13	0.00	11	0.00	0.00	0.00
	emit	25	0.00	15	0.00	0.00	0.00
	vpenta	110	8.99	82	0.39	0.00	9.38
	mxm	29	0.06	22	0.00	0.00	0.06
spice	extmem	6	1.71	21	1.77	3.41	3.47
	memadj	9	0.01	13	0.01	0.00	0.01
	find	10	0.06	60	0.46	0.00	0.51
	indxx	2	0.01	11	7.71	0.00	7.73
	reserv	6	0.57	46	4.03	0.38	4.59
	swapij	18	0.01	40	0.02	0.00	0.03
	load	149	0.00	332	0.02	0.00	0.02
	trunc	4	0.00	59	0.00	0.00	0.00
	bjt	374	9.37	538	6.19	1.14	15.56
	mcopy	1	0.00	14	0.09	0.00	0.09
	dcdcmp	8	0.00	43	0.00	0.00	0.00
tomcatv	MAIN	82	7.61	68	0.12	0.00	7.73

stores inserted for spilling, and the last column indicates that these 110 instructions account for 5.99% of the dynamically executed instructions each time this function is called. The percentage of spill code for the usage of callee-saved registers is not presented as it is a machine dependent property and therefore not calculated by the register allocator.

3.10 Macroregister Allocation

The procedure-calling convention of a particular processor architecture not only specifies the registers to be used as caller-saved and callee-saved, it also specifies the registers to be used for passing parameters during procedure calls as well as the registers used for procedure return values. These conventions must be observed when the original Lcode is being generated by the IMPACT compiler. Thus, the Lcode and subsequent Mcode will contain what is known as macroregisters. Unlike virtual registers, macroregisters have a one-to-one mapping with their corresponding registers in the register file of the target processor. In effect, we have an additional set of registers that are unavailable to the register allocator. Making these available for allocation could significantly improve the quality of the allocation.

The dataflow analysis procedure presented in Section 3.2 determines the live range for every virtual register within the function; macroregisters are ignored. It is possible to simply add these registers to the register bank definitions of the target processor, but there is a problem with this. The register allocator knows nothing of these macroregisters and blindly allocating the macroregisters to virtual registers may result in an incorrect

```
        .
        .
r1 <- r2 + r3
M0 <- r4
M1 <- r1
jsr multiply
        .
        .
```

Figure 3.13: Code sequence containing macroregisters

code sequence. To illustrate the problem, let **M**0 and **M**1 be macroregisters for the target processor. Examine the short code sequence shown in Figure 3.13. If the register allocator were to assign macroregister **M**0 to virtual register **r**1, which is certainly possible, the resulting code would be incorrect.

The solution is to enable the register allocator to determine live ranges for these macroregisters. The one obstacle to this is that the names of the macroregisters are different for each processor's code generator. The answer is to provide more register bank classes to provide the register allocator with the names of any macroregisters that may be allocated to virtual registers. The new classes are **R**_MACRO_CALLER and **R**_MACRO_CALLEE; a complete set of register bank classes is provided in Table 3.9. We have to differentiate between caller-saved and callee-saved macros, because a processor architecture may conceivably have either or both. The macroregisters of the PA-RISC would be defined as caller-saved macros, whereas the macroregisters of the Am29000 are callee-saved since they are located within a register window. Macroregisters are described to the register allocator the same way as the other registers with one small difference, the

Table 3.9: Register bank classes

Internal Type	**C**lass Description
R_CALLER	caller-saved registers
R_CALLEE	callee-saved registers
R_SPILL	spill registers
R_CALLER_MACRO	caller-saved macroregisters
R_CALLEE_MACRO	callee-saved macroregisters

provided register map contains the names of the macroregisters rather than the register name in the target processor register file.

Prior to live-range determination, any defined macroregister banks are scanned to determine the macroregisters that are available for allocation. Each of the available macroregisters is assigned a virtual register number. Then each subroutine call within the function is made to redefine all available caller-saved macroregisters and we perform reaching-definition analysis. Each subroutine call then uses only those macroregisters whose definitions actually reach that subroutine call. Now, live-variable analysis is performed as before and live ranges are constructed for the virtual registers as well as for the available macroregisters. Once a live range has been determined for each macroregister, it is simply treated during the coloring process as a virtual register that has already been assigned that particular macroregister. Thus, the only phase of register allocation that must be modified to allow macroregister allocation is the dataflow analysis phase.

3.11 Macroregister Allocation Evaluation

Performing macroregister allocation produces a better allocation, especially in the case of the PA-RISC 1.1. This is to be expected since we are increasing the number of registers available to the register allocator, and in the case of the PA-RISC 1.1 we increase the number of available caller-saved registers by seven. Thus, instead of the four caller-saved registers available to the base allocator, we now have 11. The results of macroregister allocation are provided in Table 3.10. Table 3.11 summarizes the benefits of utilizing macroregisters during allocation. In this table the **L**oads and **S**tores columns provide the change in the number of loads and stores, respectively, when macroregisters are utilized. The **%** columns following the **L**oads and **S**tores provide an estimate of the change in execution time due to the corresponding change in the number of loads and stores inserted. The final column, **T**otal %, contains an estimate of the overall change in execution time of the function.

As a result of macroallocation, the register allocator inserts an average of 23 fewer loads and 36 fewer stores into each function, although the numbers are as high as 240 fewer loads and 311 fewer stores for the **d**oduc function **d**eseco. More importantly, however, is the fact that the average execution time of these functions is decreased by 1.48% (see the last row of Table 3.11). The largest performance improvement of 7.73% occurred for the **s**pice function **i**ndxx which resulted from a savings of only 13 total spill instructions. This indicates that the amount of inserted spill code is not necessarily as important as the location of that spill code. This result is important for live-range splitting, discussed in Section 3.12, because the splitting of a live range may actually introduce more spill code than would have been inserted if the live range was spilled. As long as that spill code is less frequently executed, the splitting of the live range is beneficial.

Table 3.10: Macroregister allocation results

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**C**aller %	**T**otal %
cccp	do_define	18	1.16	20	1.45	0.66	2.61
	rescan	40	0.43	56	0.25	0.01	0.69
doduc	debflu	116	0.99	147	0.83	0.00	1.83
	debico	37	7.34	24	0.89	0.00	8.22
	deseco	255	3.31	186	1.19	0.25	4.49
	drepvi	5	0.63	8	1.01	0.00	1.64
	ihbtr	7	0.66	14	1.94	0.00	2.60
	lissag	0	0.00	0	0.00	0.00	0.00
	orgpar	34	0.01	31	1.93	0.00	1.93
	paroi	51	1.37	33	0.33	0.09	1.70
	pastem	38	0.00	74	0.02	0.01	0.03
	prophy	359	14.08	229	8.33	0.01	22.41
	repvid	11	1.54	8	0.32	0.00	1.86
	sortie	4	0.00	6	4.68	0.00	4.68
	subb	127	13.66	80	8.60	0.00	22.26
	supp	138	11.94	92	7.96	0.00	19.90
ear	EARSTEP	19	0.17	16	0.17	0.34	0.34
eqntott	read_ones	6	0.01	11	0.01	0.00	0.02
espresso	scofactor	9	0.41	16	1.17	0.00	1.58
	sccc	9	0.21	9	0.27	0.00	0.49
	tautology	11	0.32	12	0.53	0.11	0.85
fppp	fmtgen	10	1.12	13	1.46	2.01	2.57
	fpppp	2327	32.85	884	12.48	0.00	45.33
	twldrv	707	10.92	426	4.72	0.00	15.64
lex	packtrans	5	0.01	13	0.04	0.00	0.05
li	evform	51	1.27	35	2.37	1.37	3.64
	xlevlist	25	1.02	17	2.00	0.94	3.02
	xlevarg	24	0.86	16	2.03	0.02	2.89
matrix300	sgemm	2	0.04	2	0.04	0.09	0.09
	sgemv	10	0.00	10	0.00	0.00	0.00
nasa7	cfft2d1	28	0.03	19	0.00	0.00	0.03
	cfft2d2	26	0.05	16	0.00	0.00	0.05
	cholsky	56	0.11	32	0.01	0.00	0.12
	btrix	63	2.01	50	0.22	0.00	2.23
	gmtry	13	0.00	11	0.00	0.00	0.00
	emit	16	0.00	13	0.00	0.00	0.00
	vpenta	83	1.80	56	0.21	0.00	2.01
	mxm	14	0.00	11	0.00	0.00	0.01
spice	extmem	3	1.70	3	1.70	3.41	3.41
	memadj	4	0.00	4	0.00	0.00	0.00
	find	6	0.04	6	0.06	0.04	0.10
	indxx	0	0.00	0	0.00	0.00	0.00
	reserv	5	0.49	5	0.49	0.98	0.98
	swapij	0	0.00	0	0.00	0.00	0.00
	load	54	0.00	89	0.00	0.00	0.00
	trunc	4	0.00	41	0.00	0.00	0.00
	bjt	367	8.57	540	5.69	1.16	14.27
	mcopy	0	0.00	0	0.00	0.00	0.00
	dcdcmp	4	0.00	4	0.00	0.00	0.00
tomcatv	MAIN	55	3.69	42	0.06	0.00	3.75

Table 3.11: Benefit of macroregister allocation

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**T**otal %
cccp	do_define	-5	0.09	-67	-3.52	-3.47
	rescan	-13	-0.08	-45	-0.51	-0.59
doduc	debflu	-33	0.00	-92	-0.01	-0.01
	debico	-24	-3.02	-37	-0.21	-3.28
	deseco	-240	-2.76	-311	-0.91	-3.74
	drepvi	-9	-1.08	-19	-2.30	-3.40
	ihbtr	-10	0.00	-17	-0.67	-0.67
	lissag	-5	-0.01	-12	-0.01	-0.02
	orgpar	-76	-0.03	-69	-0.02	-0.05
	paroi	-50	-2.59	-93	-0.21	-2.81
	pastem	-13	0.00	-30	-0.04	-0.04
	prophy	-52	-1.91	-56	-1.13	-3.44
	repvid	-11	-3.55	-9	-0.76	-4.34
	sortie	0	0.00	-10	-0.03	-0.03
	subb	-8	-0.78	-4	-0.34	-1.27
	supp	-16	-1.24	-10	-0.73	-2.20
ear	EARSTEP	2	0.00	-9	0.00	0.00
eqntott	read_ones	0	0.00	-17	-0.03	-0.03
espresso	scofactor	-1	0.00	-12	-0.88	-0.88
	sccc	-20	-0.40	-59	-1.25	-1.65
	tautology	-3	-0.07	-23	-0.72	-0.79
fppp	fmtgen	-3	-0.01	-15	-2.10	-2.13
	fpppp	-168	-1.69	-90	-0.90	-3.51
	twldrv	-98	-2.17	-40	-0.53	-2.89
lex	packtrans	-6	-0.02	-29	-0.09	-0.11
li	evform	2	0.30	-25	-0.47	-0.17
	xlevlist	3	0.06	-10	-0.06	0.00
	xlevarg	3	0.02	-11	-0.83	-0.82
matrix300	sgemm	-14	-0.03	-3	-0.03	-0.06
	sgemv	-4	-0.04	-1	0.00	-0.04
nasa7	cfft2d1	-4	-0.01	-5	0.00	-0.01
	cfft2d2	-2	-0.01	-2	0.00	-0.01
	cholsky	-24	-0.06	-9	-0.01	-0.07
	btrix	-77	-5.44	-41	-0.90	-6.38
	gmtry	0	0.00	0	0.00	0.00
	emit	-9	0.00	-2	0.00	0.00
	vpenta	-27	-7.32	-26	-0.18	-7.52
	mxm	-15	-0.05	-11	0.00	-0.05
spice	extmem	-3	0.00	-18	-0.06	-0.07
	memadj	-5	-0.01	-9	-0.01	-0.01
	find	-4	-0.02	-54	-0.40	-0.42
	indxx	-2	-0.01	-11	-7.71	-7.73
	reserv	-1	-0.08	-41	-3.56	-3.65
	swapij	-18	-0.01	-40	-0.02	-0.03
	load	-95	0.00	-243	-0.02	-0.02
	trunc	0	0.00	-18	0.00	0.00
	bjt	-7	-0.86	2	-0.53	-1.51
	mcopy	-1	0.00	-14	-0.09	-0.09
	dcdcmp	-4	0.00	-39	0.00	0.00
tomcatv	MAIN	-27	-4.07	-26	-0.06	-4.13
AVERAGE		-23	-0.78	-36	-0.66	-1.48

In the previous discussion of live-range determination in Section 3.2, the statement was made that renumbering is not performed due to negligible benefits. Table 3.12 presents the results of applying renumbering during live range determination and Table 3.13 contains the benefits of renumbering over the benefits of macroregister allocation. Renumbering results in a performance improvement in only 6 of the 50 functions and all of the improvements are below 0.2% except for the **d**oduc function **d**ebico (0.56%). For the remaining 44 functions, half of them suffered a performance degradation and the others were unchanged. Overall, renumbering resulted in an average increase in spill code of two loads and three stores for each function and an average increase in execution time of 0.50%. These results do not necessarily place in question the use of renumbering within other allocators, instead they serve to indicate that renumbering provides little benefit under the current implementation of the IMPACT register allocator.

3.12 Live-Range Splitting

The register allocation algorithm as presented performs well, but there are instances in which a great deal of spill code is introduced. When compiling for a superscalar/VLIW processor, the goal is to achieve a high degree of instruction-level parallelism (ILP). Methods used to achieve this, such as the superblock and hyperblock scheduling models, seek to increase ILP by increasing the amount of code available to the scheduler in each control block. Certain superscalar optimizations, specifically loop unrolling, also have this goal in mind. The net result of these ILP optimizations is a dramatic increase

Table 3.12: Register allocation with renumbering

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**C**aller %	**T**otal %
cccp	do_define	35	2.09	30	2.32	0.65	4.41
	rescan	50	0.61	64	0.62	0.01	1.23
doduc	debflu	116	1.05	178	0.88	0.00	1.93
	debico	36	6.85	23	0.85	0.00	7.70
	deseco	260	3.13	184	1.24	0.25	4.38
	drepvi	8	0.99	19	2.35	0.00	3.34
	ihbtr	7	0.66	14	1.94	0.00	2.60
	lissag	0	0.00	0	0.00	0.00	0.00
	orgpar	38	0.01	33	1.93	0.00	1.93
	paroi	53	1.37	35	0.33	0.09	1.70
	pastem	30	0.00	81	0.02	0.00	0.03
	prophy	359	14.08	228	8.29	0.01	22.37
	repvid	11	1.54	8	0.32	0.00	1.86
	sortie	4	0.00	6	4.68	0.00	4.68
	subb	127	13.66	80	8.60	0.00	22.26
	supp	140	12.08	93	8.02	0.00	20.10
ear	EARSTEP	15	0.17	16	0.17	0.34	0.34
eqntott	read_ones	7	0.02	12	0.01	0.01	0.03
espresso	scofactor	16	0.41	22	1.10	0.00	1.51
	sccc	12	0.31	12	0.43	0.00	0.74
	tautology	9	0.22	11	0.44	0.00	0.66
fpppp	fmtgen	10	1.12	13	1.46	2.01	2.57
	fpppp	2340	32.93	892	12.55	0.00	45.49
	twldrv	691	11.95	410	4.80	0.00	16.74
lex	packtrans	9	0.02	17	0.06	0.00	0.08
li	evform	47	1.55	36	3.10	1.62	4.65
	xlevlist	24	1.22	18	2.23	1.45	3.45
	xlevarg	21	1.66	17	3.67	0.02	5.32
matrix300	sgemm	9	0.06	2	0.04	0.00	0.10
	sgemv	10	0.00	10	0.00	0.00	0.00
nasa7	cfft2d1	32	0.04	20	0.00	0.00	0.04
	cfft2d2	30	0.07	17	0.00	0.00	0.07
	cholsky	60	8.00	31	0.01	0.00	8.01
	btrix	83	3.81	82	2.76	0.00	6.57
	gmtry	17	0.04	12	0.00	0.00	0.04
	emit	15	0.00	12	0.00	0.00	0.00
	vpenta	80	1.77	54	0.19	0.00	1.97
	mxm	10	0.00	8	0.00	0.00	0.00
spice	extmem	3	1.70	3	1.70	3.41	3.41
	memadj	4	0.00	4	0.00	0.00	0.00
	find	6	0.04	6	0.06	0.04	0.10
	indxx	0	0.00	0	0.00	0.00	0.00
	reserv	7	0.68	7	0.68	1.37	1.37
	swapij	0	0.00	0	0.00	0.00	0.00
	load	82	0.00	138	0.00	0.00	0.00
	trunc	11	0.00	57	0.00	0.00	0.00
	bjt	368	10.34	533	5.60	0.32	15.94
	mcopy	0	0.00	0	0.00	0.00	0.00
	dcdcmp	6	0.00	6	0.00	0.00	0.00
tomcatv	MAIN	54	3.69	41	0.06	0.00	3.75

Table 3.13: Benefit of register allocation with renumbering

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**T**otal %
cccp	do_define	17	0.94	10	0.89	1.88
	rescan	10	0.18	8	0.37	0.55
doduc	debflu	0	0.06	31	0.05	0.11
	debico	-1	-0.52	-1	-0.04	-0.56
	deseco	5	-0.18	-2	0.06	-0.12
	drepvi	3	0.36	11	1.37	1.76
	ihbtr	0	0.00	0	0.00	0.00
	lissag	0	0.00	0	0.00	0.00
	orgpar	4	0.00	2	0.00	0.00
	paroi	2	0.00	2	0.00	0.00
	pastem	-8	0.00	7	0.00	0.00
	prophy	0	0.01	-1	-0.04	-0.04
	repvid	0	0.00	0	0.00	0.00
	sortie	0	0.00	0	0.00	0.00
	subb	0	0.00	0	0.00	0.00
	supp	2	0.16	1	0.07	0.26
ear	EARSTEP	-4	0.00	0	0.00	0.00
eqntott	read_ones	1	0.00	1	0.00	0.01
espresso	scofactor	7	0.00	6	-0.07	-0.07
	sccc	3	0.10	3	0.16	0.25
	tautology	-2	-0.10	-1	-0.09	-0.19
fpppp	fmtgen	0	0.00	0	0.00	0.00
	fpppp	13	0.13	8	0.09	0.30
	twldrv	-16	1.16	-16	0.08	1.32
lex	packtrans	4	0.02	4	0.02	0.03
li	evform	-4	0.29	1	0.76	1.06
	xlevlist	-1	0.20	1	0.23	0.44
	xlevarg	-3	0.81	1	1.70	2.57
matrix300	sgemm	7	0.02	0	0.00	0.02
	sgemv	0	0.00	0	0.00	0.00
nasa7	cfft2d1	4	0.01	1	0.00	0.01
	cfft2d2	4	0.02	1	0.00	0.02
	cholsky	4	8.58	-1	0.00	8.58
	btrix	20	1.88	32	2.62	4.65
	gmtry	4	0.03	1	0.00	0.03
	emit	-1	0.00	-1	0.00	0.00
	vpenta	-3	-0.03	-2	-0.02	-0.04
	mxm	-4	0.00	-3	0.00	0.00
spice	extmem	0	0.00	0	0.00	0.00
	memadj	0	0.00	0	0.00	0.00
	find	0	0.00	0	0.00	0.00
	indxx	0	0.00	0	0.00	0.00
	reserv	2	0.19	2	0.19	0.39
	swapij	0	0.00	0	0.00	0.00
	load	28	0.00	49	0.00	0.00
	trunc	7	0.00	16	0.00	0.00
	bjt	1	1.97	-7	-0.10	1.99
	mcopy	0	0.00	0	0.00	0.00
	dcdcmp	2	0.00	2	0.00	0.00
tomcatv	MAIN	-1	0.00	-1	0.00	0.00
AVERAGE		2	0.33	3	0.17	0.50

in register pressure. The higher register pressure results in more spill code. In some cases, the spill code may be more than can be absorbed by the high issue rate of the processor, degrading the performance of the code, possibly negating any benefit gained by application of the optimization in the first place.

One method to reduce the amount spill code is to break up the live range of an unallocatable virtual register and allocate as much of it as possible. Ideally, breaking up the live range and allocating portions of it will result in less spill code. Chow and Hennessy [4] were the first to propose their approach, termed live range splitting. A brief description of their method was presented in Section 2.2.3. The many problems involved in live-range splitting, including the selection of live ranges to split and selecting the split points within the live range, are not trivial. An excellent discussion of these problems and the many issues involved as well as several methods of live-range splitting, may be found in P. Briggs' thesis [5]. One of these, loop-level live-range splitting, has been implemented within the IMPACT register allocator.

If loop-level splitting fails to produce an allocatable live range, the register allocator may attempt instruction-level splitting. The instruction-level splitting algorithm currently being implemented within the IMPACT register allocator is very similar to the method proposed by Chow and Hennessy [4]. As the implementation is not yet complete and the algorithm is still evolving as implementation proceeds, it will not be discussed further.

3.12.1 Loop splitting

Since the register pressure problem is severe within loops, loop-level live-range splitting is a good starting point on the path to instruction-level live-range splitting. The goal of loop-level splitting is to relieve the register pressure within loops by removing interferences caused by live ranges that are inactive within the loop. Also, loop-level splitting may be used to reduce the number of interferences of live range containing a loop with high register pressure, by removing that loop from the live range and making the live range allocatable in the process.

Accommodating live-range splitting within the existing register allocation algorithm is relatively straightforward. Rather than simply spilling an unallocatable virtual register, the allocator attempts loop-level live-range splitting in an effort to allocate some portion of the virtual register's live range. The obvious first step is to examine the live range to locate and remove any loops contained therein. If a live range is found to contain one or more loops, the live-range instructions contained within those loops and the loops' preheader blocks are removed from the original live range. If on the other hand, the live range contains no loops, the virtual register is spilled. The loop removal algorithm is shown in Figure 3.14.

Consider the live range in Figure 3.15(a) which contains three loops, **A**, **B**, **C**. The blocks labeled **A**p, **B**p, and **C**p represent the preheader block of each loop. Performing loop removal on these live ranges results in the live range in Figure 3.15(b). The uncolored

```
base = lowest nesting level of live range
for ( i = base+1; i <= base+2; i++ )
    for ( each loop at nesting level i )
        if ( live_range_instr ∩ loop_instr ≠ ∅ ) {
            live_range_instr = live_range_instr - loop_instr
            live_range_instr = live_range_instr - preheader_instr
        }
```

Figure 3.14: Loop removal algorithm

blocks are those blocks no longer contained within the live range. Figure 3.15(c) contains the three removed loops which themselves may become individual live ranges.

Once all loops have been removed from the current live range, the live-range's interferences are updated and the register allocator checks for available free registers. In the event that no register is available, the unallocatable portion of the live range is spilled. In our example, the portion of the live range in Figure 3.15(b) would be spilled. The removed loops become new live ranges and are added to the heap for future consideration. If a free register is available, the register allocator attempts to reinsert the loops in order to allocate as much of the live range as possible. Each removed loop is assigned a priority value, calculated from Eq. (3.5), and reinsertion is attempted starting with the loop of highest priority.

$$LP(loop) = \frac{\sum_{i \in loop}(def_i + use_i) \times w_i}{N} \tag{3.5}$$

After a loop has been reinserted, the live-range's allocatability is checked again. If the live range is allocatable, the next loop is reinserted and the allocatability of the live range is reexamined. Any loop that causes the live range to become no longer allocatable is

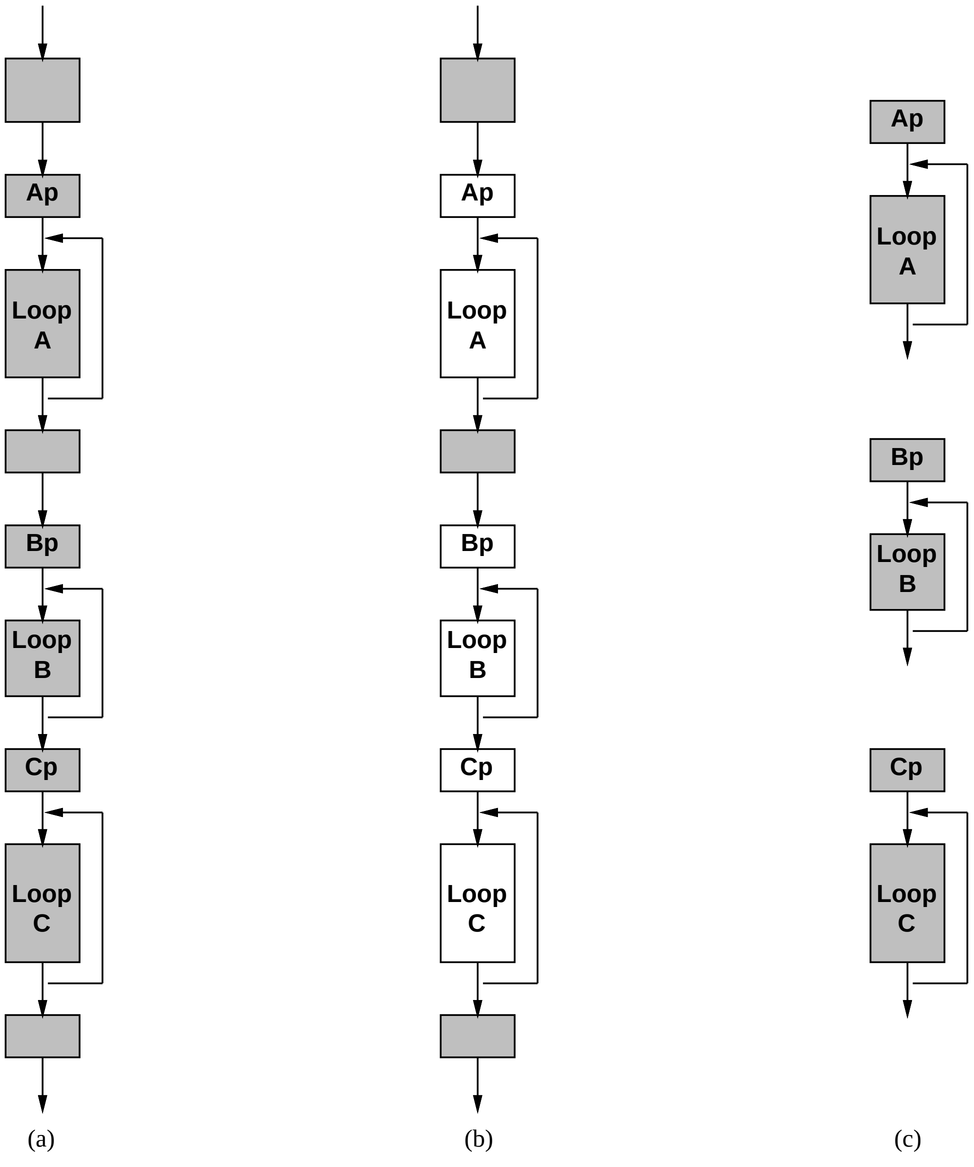

Figure 3.15: Loop removal example: (a) original live range, (b) live range after removal, (c) new loop live ranges

```
for ( each removed loop )
    < calculate loop priority function >
< sort loops by priority >
for ( each removed loop in priority order )
    live_range_instr = live_range_instr ∪ loop_instr
    if ( live range unallocatable )
        live_range_instr = live_range_instr - loop_instr
```

Figure 3.16: Loop reinsertion algorithm

removed once again. This process continues until an attempt has been made to reinsert each loop. The loop reinsertion algorithm is shown in Figure 3.16.

For example, assume that the loops of Figure 3.15(c) have a priority order of **B**, **C**, **A**, loop **B** is reinserted into the live range of Figure 3.15(b), and the live range is found to be allocatable, resulting in Figure 3.17(a). Subsequent attempts to reinsert **C** and then **A** both result in unallocatability and those loops become candidates for future allocation. Thus in this example, the portion of the live range that is finally allocated is shown in Figure 3.17(a).

3.12.2 Correction code

An allocated live range that has been split in this manner is no longer a valid live range and additional instructions are required for correctness. Something must be done about the gaps in the allocated live range in Figure 3.17(a). However, no correction code is inserted at the time this live range is allocated. The reason is that the final form of the required correction code depends upon the allocation results of any removed loops. The

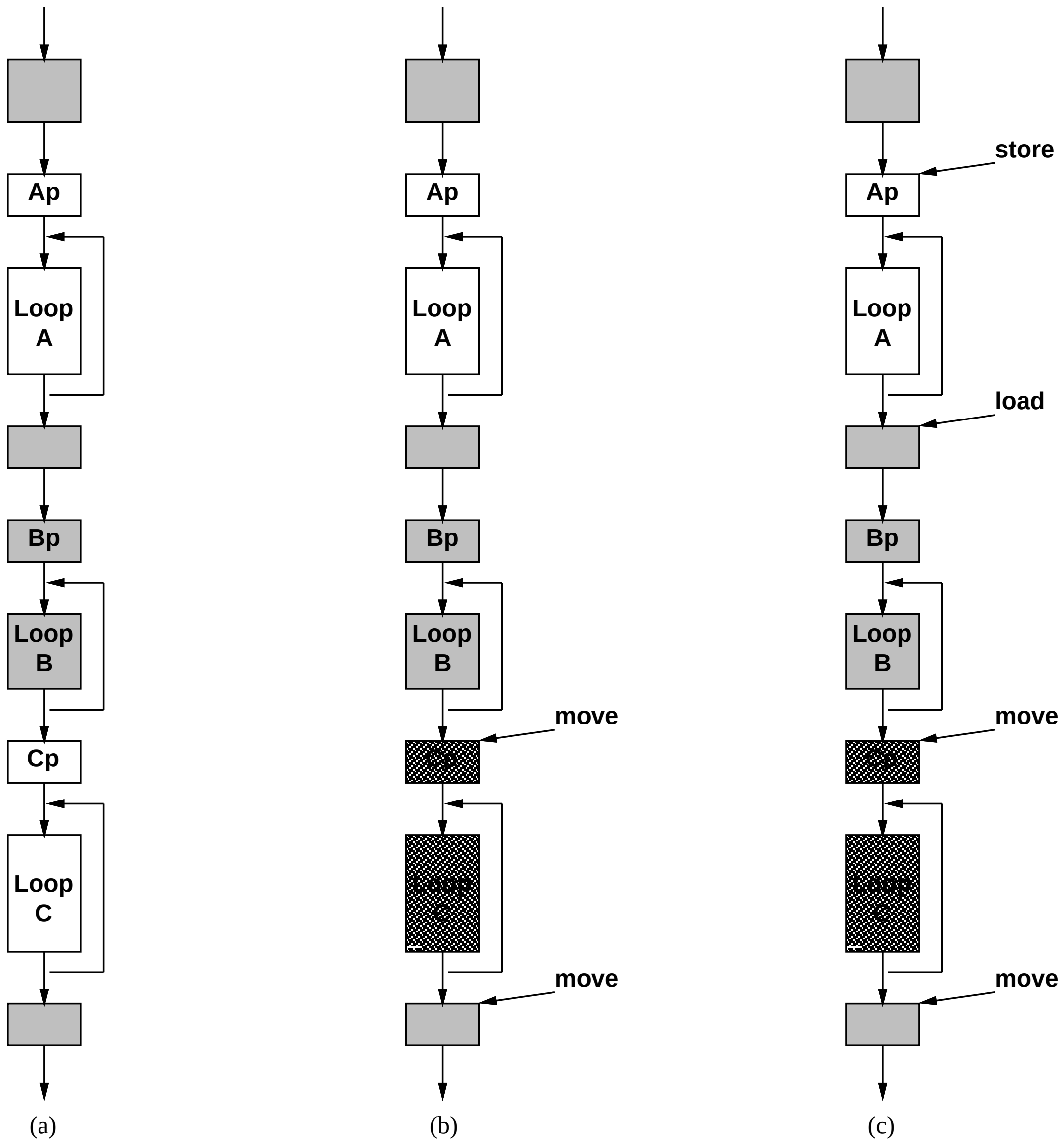

Figure 3.17: Loop splitting correction code: (a) Loop B is reinserted, (b) Loop C is allocated requiring the indicated **m**ov instructions, (c) Loop A is spilled requiring the indicated **l**oad and **s**tore instructions

four possible cases are described below. For the purposes of this discussion, the original live range from which a loop was extracted is termed the **p**arent of that loop.

In the case in which the parent live range is allocated, as in our previous example, the removed loops may be either allocated or spilled. The preferred case is allocation of the removed loop. In this event, **m**ove instructions are required at the entry and exit blocks of the loop to place the value in the proper register. If we assume that this is the case for loop **C** in our running example, **m**ove instructions are required as indicated in Figure 3.17(b). There are two instances where these instructions are unnecessary: when the first reference in the loop live range is a definition and when the value is not live upon exit from the loop. Spilling a removed loop requires insertion of **s**tore and **l**oad instructions to save the value upon entry and restore it upon exit. If we assume that loop **A** is spilled, **s**tore and **l**oad instructions are required as shown in Figure 3.17(c). These instructions may also be unnecessary for the same reasons mentioned earlier. The third case involves the spilling of the parent live range and the allocation of a removed loop. Here a **l**oad is required upon entry and a store is required upon exit. The final case in which both are spilled obviously does not require additional instructions for correctness.

3.12.3 Loop splitting evaluation

The allocation results for loop-level live-range splitting are provided in Table 3.14 and the benefit of loop-level splitting over macroregister allocation is provided in Table 3.15. The results show a minimal amount of performance improvement. Although loop-level splitting was able to achieve a decrease in the amount of spill code for all cases in which a change occurred, the percent improvement in execution time in those cases is rather small. There are a number of possible explanations for these results. First, in the cases in which there was a large decrease in the static spill code count, the dynamic spill percentages are about 2-3% after macroregister allocation, and thus so is the maximum achievable performance improvement. Furthermore, a number of the functions in which no improvement was realized simply contained a limited number of loops. Also, the granularity of splitting at the loop level may be too large to exploit all of the potential gains that may be achieved from live-range splitting. Overall, loop-level live-range splitting was consistently able to reduce the amount of spill code inserted by the register allocator, and since the increase in allocation time is minimal, loop-level splitting may become a viable method for reducing spill code.

Table 3.14: Allocation results with loop splitting

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**C**aller %	**T**otal %
cccp	do_define	13	0.76	20	1.28	0.66	2.04
	rescan	21	0.20	58	0.22	0.01	0.42
doduc	debflu	94	0.67	145	0.52	0.00	1.20
	debico	35	7.34	23	0.89	0.00	8.22
	deseco	249	3.30	184	1.18	0.25	4.48
	drepvi	5	0.63	8	1.01	0.00	1.64
	ihbtr	3	0.21	11	0.94	0.00	1.15
	lissag	0	0.00	0	0.00	0.00	0.00
	orgpar	34	0.01	31	1.93	0.00	1.93
	paroi	31	0.05	24	0.15	0.10	0.20
	pastem	38	0.00	74	0.02	0.01	0.03
	prophy	331	13.31	213	8.32	0.01	21.63
	repvid	11	1.54	8	0.32	0.00	1.86
	sortie	4	0.00	6	4.68	0.00	4.68
	subb	127	13.66	80	8.60	0.00	22.26
	supp	138	11.94	92	7.96	0.00	19.90
ear	EARSTEP	19	0.17	16	0.17	0.34	0.34
eqntott	read_ones	6	0.01	11	0.01	0.00	0.02
espresso	scofactor	6	0.41	15	1.17	0.00	1.58
	sccc	9	0.21	9	0.27	0.00	0.49
	tautology	11	0.32	12	0.53	0.11	0.85
fpppp	fmtgen	10	1.12	13	1.46	2.01	2.57
	fpppp	2327	32.85	884	12.48	0.00	45.33
	twldrv	571	10.88	301	4.70	0.03	15.65
lex	packtrans	4	0.00	12	0.04	0.00	0.04
li	evform	30	1.03	26	1.62	1.39	2.65
	xlevlist	14	0.49	12	1.01	0.96	1.50
	xlevarg	14	0.86	12	1.46	0.02	2.32
matrix300	sgemm	2	0.04	2	0.04	0.09	0.09
	sgemv	10	0.00	10	0.00	0.00	0.00
nasa7	cfft2d1	9	0.00	10	0.00	0.00	0.00
	cfft2d2	6	0.00	6	0.00	0.00	0.00
	cholsky	31	0.03	24	0.00	0.00	0.03
	btrix	24	1.33	32	0.20	0.00	1.54
	gmtry	0	0.00	0	0.00	0.00	0.00
	emit	3	0.00	3	0.00	0.00	0.00
	vpenta	28	0.21	25	0.13	0.00	0.34
	mxm	0	0.00	0	0.00	0.00	0.00
spice	extmem	3	1.70	3	1.70	3.41	3.41
	memadj	4	0.00	4	0.00	0.00	0.00
	find	6	0.04	6	0.06	0.04	0.10
	indxx	0	0.00	0	0.00	0.00	0.00
	reserv	5	0.49	5	0.49	0.98	0.98
	swapij	0	0.00	0	0.00	0.00	0.00
	load	54	0.00	89	0.00	0.00	0.00
	trunc	4	0.00	41	0.00	0.00	0.00
	bjt	367	8.57	540	5.69	1.16	14.27
	mcopy	0	0.00	0	0.00	0.00	0.00
	dcdcmp	4	0.00	4	0.00	0.00	0.00
tomcatv	MAIN	23	1.28	19	0.04	0.00	1.32

Table 3.15: Benefit of loop splitting

Benchmark	**F**unction	**L**oads	%	**S**tores	%	**T**otal %
cccp	do_define	-5	-0.41	0	-0.17	-0.58
	rescan	-19	-0.24	2	-0.03	-0.27
doduc	debflu	-22	-0.32	-2	-0.31	-0.64
	debico	-2	0.00	-1	0.00	0.00
	deseco	-6	0.00	-2	-0.01	-0.01
	drepvi	0	0.00	0	0.00	0.00
	ihbtr	-4	-0.44	-3	-1.01	-1.46
	lissag	0	0.00	0	0.00	0.00
	orgpar	0	0.00	0	0.00	0.00
	paroi	-20	-1.31	-9	-0.19	-1.50
	pastem	0	0.00	0	0.00	0.00
	prophy	-28	-0.88	-16	-0.01	-0.98
	repvid	0	0.00	0	0.00	0.00
	sortie	0	0.00	0	0.00	0.00
	subb	0	0.00	0	0.00	0.00
	supp	0	0.00	0	0.00	0.00
ear	EARSTEP	0	0.00	0	0.00	0.00
eqntott	read_ones	0	0.00	0	0.00	0.00
espresso	scofactor	-3	0.00	-1	0.00	0.00
	sccc	0	0.00	0	0.00	0.00
	tautology	0	0.00	0	0.00	0.00
fpppp	fmtgen	0	0.00	0	0.00	0.00
	fpppp	0	0.00	0	0.00	0.00
	twldrv	-136	-0.05	-125	-0.03	0.01
lex	packtrans	-1	0.00	-1	0.00	0.00
li	evform	-21	-0.23	-9	-0.77	-1.01
	xlevlist	-11	-0.53	-5	-1.00	-1.55
	xlevarg	-10	0.00	-4	-0.58	-0.58
matrix300	sgemm	0	0.00	0	0.00	0.00
	sgemv	0	0.00	0	0.00	0.00
nasa7	cfft2d1	-19	-0.02	-9	0.00	-0.03
	cfft2d2	-20	-0.04	-10	0.00	-0.05
	cholsky	-25	-0.08	-8	-0.01	-0.09
	btrix	-39	-0.69	-18	-0.01	-0.70
	gmtry	-13	0.00	-11	0.00	0.00
	emit	-13	0.00	-10	0.00	0.00
	vpenta	-55	-1.59	-31	-0.08	-1.67
	mxm	-14	0.00	-11	0.00	-0.01
spice	extmem	0	0.00	0	0.00	0.00
	memadj	0	0.00	0	0.00	0.00
	find	0	0.00	0	0.00	0.00
	indxx	0	0.00	0	0.00	0.00
	reserv	0	0.00	0	0.00	0.00
	swapij	0	0.00	0	0.00	0.00
	load	0	0.00	0	0.00	0.00
	trunc	0	0.00	0	0.00	0.00
	bjt	0	0.00	0	0.00	0.00
	mcopy	0	0.00	0	0.00	0.00
	dcdcmp	0	0.00	0	0.00	0.00
tomcatv	MAIN	-32	-2.44	-23	-0.02	-2.47
AVERAGE		-10	-0.19	-6	-0.08	-0.27

4. CONCLUSIONS AND FUTURE WORK

This thesis presents a discussion of the implementation of a global graph coloring register allocator within the framework of the IMPACT-I C compiler based upon the work done by Chaitin and Chow. The machine independent interface allows the register allocator to be used to study a large variety of register file configurations. This ability also makes possible the seamless integration of the register allocator with our current and any future processor code generators. Results show that by making macroregisters available to the register allocator, it is possible to improve the quality of the resulting allocation. Also, it has been shown that loop-level live-range splitting can consistently reduce the amount of spill code.

There are several areas within the current implementation of the register allocator that may be enhanced in order to improve the allocation quality as well as the speed of the allocation. The most important area in terms of allocation time is dataflow analysis. The current method of instruction-level dataflow analysis, although exact, is slow for large functions, and therefore demands a more efficient method. Since the current implementation of the interference graph requires a large amount of memory, a better implementation has been devised to reduce the memory requirement while maintaining the current flexibility and improving the access time.

Register allocation quality can be improved by altering the way the current register allocator handles spill registers. Instead of reserving a certain number for spilling during the allocation, the register allocator can deal with spilling as is done in Chaitin's allocator. Although results that have been achieved for loop splitting indicate that we can consistently reduce the amount of spill code, more work needs to be done to improve the reduction and to investigate the limit of what can be achieved by splitting at such a high level. The completion of the instruction-level live-range splitting implementation should complement the existing loop-level splitting process and improve our live-range splitting results. Finally, work is underway to incorporate constant preloading within the current register allocation framework and to investigate the benefits that can be achieved by performing constant preloading during register allocation.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[2] R. L. Sites and D. R. Perkins, "Machine-independent register allocation," in *Proceedings of the ACM SIGPLAN 79 Symposium on Compiler Construction*, pp. 221–225, August 1979.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer Languages*, vol. 6, pp. 47–57, January 1981.

[4] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 501–536, October 1990.

[5] P. Briggs, *Register Allocation via Graph Coloring.* Ph.D. dissertation, Department of Computer Science, Rice University, Houston, TX, 1992.

[6] D. Callahan and B. Koblenz, "Register allocation via hierarchical graph coloring," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pp. 192–203, June 1991.

[7] F. Chow, "Minimizing register usage penalty at procedure calls," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 85–93, June 1989.

[8] D. W. Wall, "Global register allocation at link time," in *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pp. 264–275, June 1986.

[9] V. Santhanam and D. Odnert, "Register allocation across procedure and module boundaries," in *Proceedings of the ACM SIGPLAN 90 Symposium on Programming Language Design and Implementation*, pp. 40–52, June 1990.

[10] R. Sethi, "Complete register allocation problems," *SIAM Journal on Computing*, vol. 4, pp. 226–248, March 1975.

[11] M. A. Auslander and M. E. Hopkins, "An overview of the PL.8 compiler," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 22–31, June 1982.

[12] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *Proceedings of the ACM SIGPLAN 82 Symposium on Compiler Construction*, pp. 98–105, June 1982.

[13] P. Briggs, K. D. Cooper, K. Kennedy, and L. Torczon, "Coloring heuristics for register allocation," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 275–284, June 1989.

[14] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pp. 311–321, June 1992.

[15] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Pinter, "Spill code minimization techniques for optimizing compilers," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, pp. 258–263, June 1989.

[16] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pp. 222–232, June 1984.

[17] J. R. Larus and P. N. Hilfinger, "Register allocation in the spur lisp compiler," in *Proceedings of the ACM SIGPLAN 86 Symposium on Compiler Construction*, pp. 255–263, June 1986.

[18] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1992.

[19] H.-P. Co., *PA-RISC 1.1 Architecture and Instruction Set Reference Manual.* Cupertino, CA: Hewlett-Packard Co., 1990.

[20] H.-P. Co., *PA-RISC Procedure Calling Conventions Reference Manual.* Cupertino, CA: Hewlett-Packard Co., 1991.

[21] M. Johnson, *Am29000 User's Manual.* Sunnyvale, CA: Advanced Micro Devices, Inc., 1990.

[22] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective structure for VLIW and superscalar compilation," tech. rep., Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, February 1992.

[23] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992.

[24] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, MA: Addison-Wesley, 1988.

[25] J. R. Goodman and W. C. Hsu, "Code scheduling and register allocation in large basic blocks," in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

APPENDIX A PA-RISC 1.1 AND AM29000 REGISTER MAPS

```
int caller_int_map[] = {
      19, 20, 21, 22
};

int callee_int_map[] = {
      3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
};

int caller_double_map[] = {
      8, 8+32, 9, 9+32, 10, 10+32, 11, 11+32, 22, 22+32, 23, 23+32,
      24, 24+32, 25, 25+32, 26, 26+32, 27, 27+32, 28, 28+32, 29, 29+32,
      30, 30+32, 31, 31+32,
};

int callee_double_map[] = {
      12, 12+32, 13, 13+32, 14, 14+32, 15, 15+32, 16, 16+32, 17, 17+32,
      18, 18+32, 19, 19+32, 20, 20+32, 21, 21+32,
};
```

Figure A.1: PA-RISC 1.1 register maps

The register maps provided to the register allocator are generated directly from the register file of the target processor. Figure A.1 shows the register maps for the PA-RISC 1.1. Note that there are only four maps for the six register banks in the register file description. The two arrays **c**aller_int_map and **c**allee_int_map are derived directly from

Table 3.3. The **c**aller_int_map array contains the four available caller-saved registers, gr19-gr22, and the **c**allee_int_map array contains the 16 available callee-saved registers, gr3-gr18. The remaining two arrays **c**aller_double_map and **c**allee_double_map are used for both the single-precision and the double-precision register banks, as they are the same physical register.

When the array **c**aller_double_map is used as a map for the double-precision, caller-saved register bank, the only values seen by the register allocator are those in odd-numbered locations, containing registers fr8-fr11 and fr22-fr31, because double-precision registers are described as having a size of 2. However, when used as a map for the single-precision, caller-saved register bank, every location is seen by the allocator since the single-precision registers have a size of 1. A double-precision register in the PA-RISC, for example **f**r8, may also serve as two single-precision registers, denoted by **f**r8L for the left half and **f**r8R for the right half. Thus, in the map, **f**r8L is denoted by 8 in the first location and **f**r8R is denoted by 8+32. The number 32 is added to the entries corresponding to right halves so that each map entry contains a unique number. The same holds true for the array **c**allee_double_reg_map[].

In the case of the Am29000, there are only two arrays (see Figure A.2), since integer, single-precision, and double-precision data all reside in the same physical register file. The **c**aller_reg_map array contains the 19 available caller-saved registers, gr98-gr111 and

```
int caller_reg_map[] = {
      98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
      116, 117, 118, 119, 120
};

int callee_reg_map[] = {
      130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
      144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157,
      158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171,
      172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185,
      186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199,
      200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213,
      214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227,
      228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241,
      242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255
};
```

Figure A.2: Am29000 register maps

gr116-gr120, and the **c**allee_reg_map array contains the 126 available callee-saved registers, fr2-fr128. There are no left and right halves to worry about, as in the PA-RISC, so these maps are derived directly from Table 3.5.