Global Reduction of Spill Code by Live Range Splitting

By
Thomas R. Suchyta

A THESIS
Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

MICHIGAN TECHNOLOGICAL UNIVERSITY
1998

This thesis, "Global Reduction of Spill Code by Live Range Splitting", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT of Computer Science

Thesis Advisor	Dr. Steve Carr
Thesis Advisor	Dr. Philip Sweany
Head of Department	Dr. Linda Ott
Date	

Abstract

Current, state-of-the-art compilers use Briggs' graph coloring heuristic for register allocation. This heuristic provides an efficient mapping of program variables to machine registers. However, if a variable cannot be assigned a register the variable is spilled and referenced through memory. Each use of the variable is preceded by a load from memory and each definition is followed by a store to memory. The algorithm presented in this thesis is a method to reduce the amount of spill code added by a Briggs' allocator.

Graph coloring maps the live range of a variable to a machine register. If no machine register is available for a live range, the variable is spilled. There are often areas in the live range where spill code is not needed. Our algorithm identifies there areas, known as low register pressure regions. Once low register pressure regions are found for a spilled live range, it is possible to limit the amount of spill code added to the low register pressure region. This is known as live range splitting. This step can be done without substantially increasing compile times since we base computing low register pressure regions on Reif and Tarjan's near linear algorithm for computing symbolic covers. The research done for this thesis shows that up to 25% of dynamic loads can be reduced with an up to 20% reduction in execution time and an average 2.3% increase register allocation time.

Acknowledgments

I would like to thank the support, advice, and guidance from my co-advisors, Dr. Carr and Dr. Sweany. With their encouragement, I've followed a challenging path for which I will always be grateful. I would also like to thank the rest of my committee, Dr. Poplawski and Dr. Gilpin, for their support.

I need to thank my family, especially my parents, Richard and Veronica Suchyta. Finally, I need to thank Janice, without whom this would have not been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One goal of an optimizing compiler is to keep frequently used program variables in machine registers rather than have the variables stored in memory. The main reason for this is the current differences between CPU cycle times and memory cycle times. If a variable must be accessed through memory, this may create a substantial number of idle CPU cycles. To avoid these long memory latencies, the compiler must find a way to keep as many variables in registers as possible.

There are two stages of a compiler that occur when program variables are mapped to machine registers. The register allocation step determines the set of variables that can be allocated to machine registers. The register assignment step then maps those variables to particular registers. For this discussion, the term register allocation will refer to both steps. It is often impossible, however, to assign each variable to a physical register. In many cases, there will be more variables than available registers. It is then necessary to access these variables through memory. These accesses are known as *spill code*. Each definition of a spilled variable is followed by a store to memory. Each use of a spilled variable is preceded by a load from memory. The algorithm presented in my thesis is a method that can reduce the amount of spill code generated by the compiler. This algorithm identifies areas in programs where a store may not be needed following a definition of a spilled variable or an area where a load may not be needed prior to a use of a spilled variable.

In order to identify areas in programs where spill code may not be needed, it is necessary to examine how the register allocator determines which definitions and uses of a variable are eligible for the same machine register. Current register allocators assign registers to the *live range* of a variable. The live range is the range from the initial definition of a variable to its final use. If there is a register available for the live range, each use and definition of the variable in the live range can be replaced with that specific machine register. If a register is not available for a live range, meaning the live range is spilled, spill code is generated for each definition and each use of the variable within the live range.

By adding spill code for each definition and each use in a spilled live range, the register allocator uses a *spill-everywhere* approach. This spill-everywhere approach is the method used by most current register allocators. The reason for adding these additional memory references to the program is a result of using a graph coloring algorithm that most current compilers use for register allocation. As will be discussed, graph coloring is an efficient technique for mapping variables to registers. However, when variables are chosen to spill, the graph coloring algorithm does not take into account that there may be certain areas in the spilled live range where spill code is not needed. In order to find these regions, it is necessary to identify the reason the live range could not be allocated a register. At some point in a spilled live range, the number of live ranges that are also live at that point must be greater than the number of available machine registers. For example, consider the following simple statement:

$$A = B + C$$

If there are uses of variables A,B, and C following this statement, the live ranges for A, B, and C *conflict* at this point. Therefore, this statement requires 3 machine registers. If there are only 2 available machine registers, it will be necessary to spill one of the variables.

The number of live ranges that conflict at any point in a program is known as *register pressure.* As shown above, if the register pressure, at some point in a program, is greater than the number of available machine registers, one or more of the live ranges must spill. However, throughout the entire spilled live range, the register pressure may not always be greater than the number of machine registers. By identifying these *low register pressure*

regions, it can be determined whether spill code is needed in these regions. If a register allocator does not generate spill code in a low register pressure region, this is known as *live range splitting.*

Our algorithm uses live range splitting by finding low register pressure regions and limiting the amount of spill code added to those regions. This approach was designed with several goals. One goal was that live range splitting be able to be easily incorporated into the stages of the current, state-of-the-art technique of register allocation. This type of register allocator, which uses a graph coloring algorithm, is based on research done by Briggs[3, 4, 5]. It will later be shown how our algorithm is incorporated into the steps of a Briggs' allocator. A second goal to our live range splitting algorithm is that the extra work required to split live ranges does not substantially increase compile times. To obtain this goal, the algorithm for finding low register pressure regions is based on the work done by Reif and Tarjan for finding symbolic covers[17]. The Reif and Tarjan algorithm has an almost linear running time and we felt this efficiency would prevent substantial increases in compile times when live range splitting was performed. The research performed for this thesis shows that these goals were obtained through a reduction in the amount of spill code added to the program without a large penalty in compile times. For the programs tested there was up to a 20% reduction in execution time through a 25% reduction in the number of dynamic memory references. The average increase in register allocation compile time was 2.3% with a decrease of 10% in register allocation time in the best case.

1.1 Example

The concept of live range splitting can be shown by the following example. Figure 1.1 shows a code fragment and the corresponding live ranges for the five variables. For this example, we assume there are 2 available machine registers. Since the number of conflicting live ranges is 3 at some point in Figure 1.1, one of the live ranges must spill. A register allocator will try to choose a variable that would be the cheapest to spill. This means a variable that would add the minimum amount of spill code or spill code that is added to an area of code that executes infrequently. For these reasons, variable A is chosen to

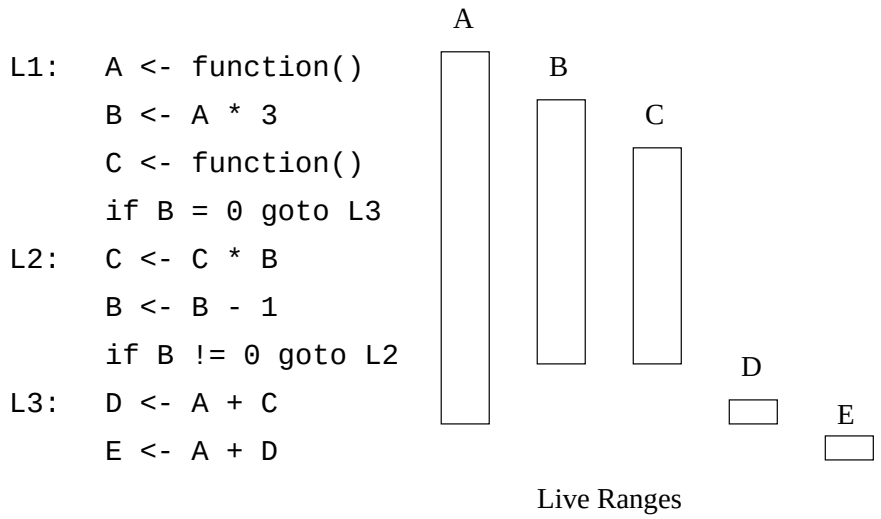

Figure 1.1: Example of live ranges.

spill. Figure 1.2 shows the code after A spills using a spill-everywhere approach. There are regions in variable A's live range where the register pressure is equal to the number of machine registers. This is shown in Figure 1.3.

Thus, if we split the live range of variable A, we can limit the amount of spill code that is generated in the low register pressure regions. In this example, it is necessary to add a store of A to memory at the exit point of the first split of A. It is then necessary to reload the variable back into a register at the second split point of A. These split points, and the spill code generated from live range splitting, are shown in Figure 1.4. Using live range splitting, we are able to reduce the number of loads from 3 to 1.

```
L1:   A <- function()
      store A
      load A
      B <- A * 3
      C <- function()
L2:   if B = 0 goto L3
      C <- C * B
      B <- B - 1
      if B != 0 goto L2
L3:   load A
      D <- A + C
      load A
      E <- A + D
```

Figure 1.2: Example of spill-everywhere approach.

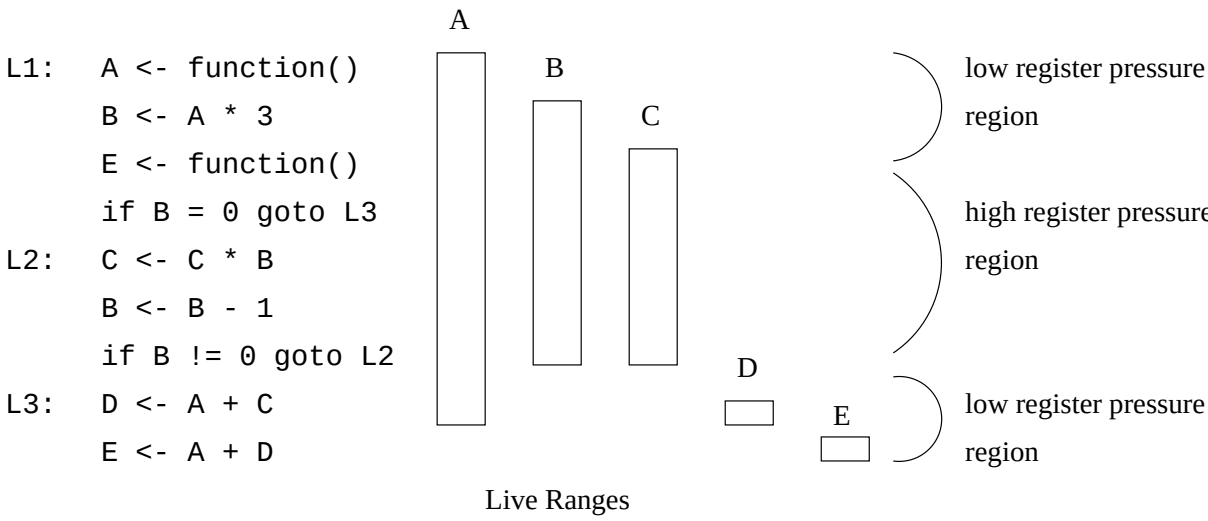

Figure 1.3: Example of low register pressure regions.

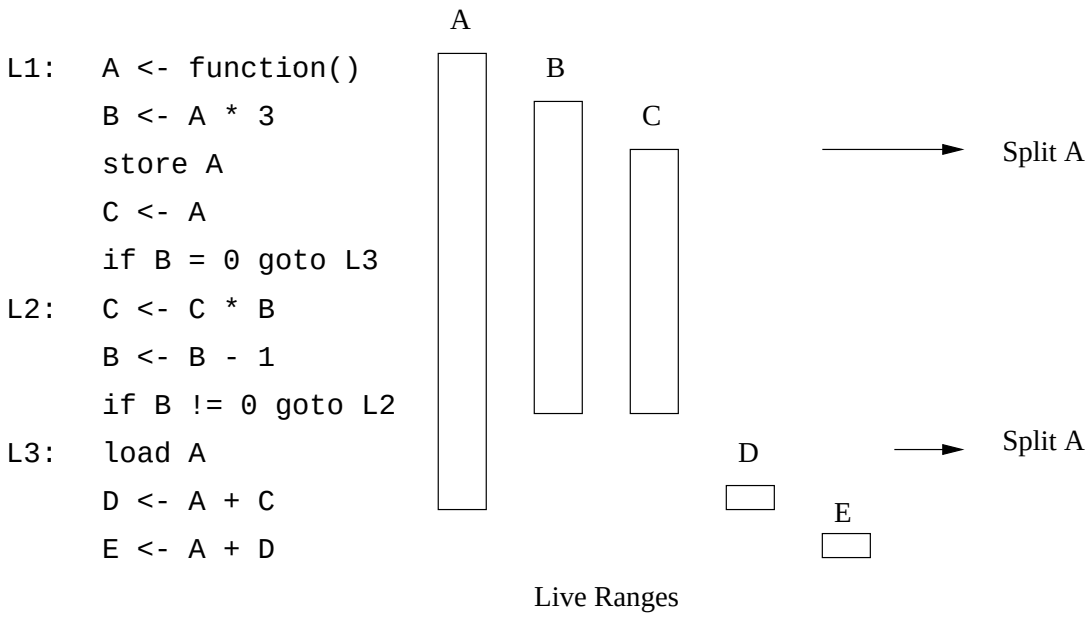

Figure 1.4: Example of live range splitting.

Chapter 2

Background on Register Allocation

As previously mentioned, current compilers map program variables to machine registers by modeling the register allocation problem as a graph coloring problem. The following is a discussion on how this is accomplished. The first implementation of register allocation by graph coloring was done by Chaitin[7, 8].

The first step in using graph coloring to perform register allocation is to determine which variables can be assigned registers. In other words, each live range must be identified. A graph is then built such that the nodes represent the live ranges and the edges represent the conflicts between those live ranges. This graph is known as an *interference graph* or *conflict graph.* The colors represent the number of machine registers. The graph is colored by finding a color for each node such that one node's color is not shared by any of its adjacent nodes. Therefore, assigning machine registers to live ranges becomes the problem of finding a k-coloring for the interference graph, where k is the number of machine registers. Since this problem is NP-complete[14], heuristics are needed to find a k-coloring.

Chaitin's heuristic is based on the following property of the interference graph. If a node in the graph has a degree $<$ k, the graph is k-colorable, if and only if, the graph is k-colorable with that node and its adjacent edges removed from the graph[7, 8]. These *unconstrained* nodes can be removed from the graph which will hopefully unconstrain other nodes in the graph that initially had degrees $\geq$ k. If an empty graph is eventually obtained, the graph

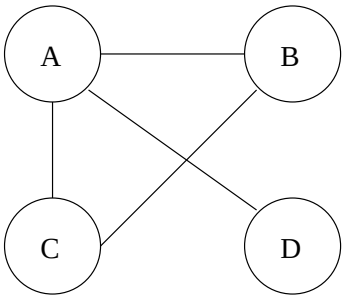

Figure 2.1: Example of an interference graph.

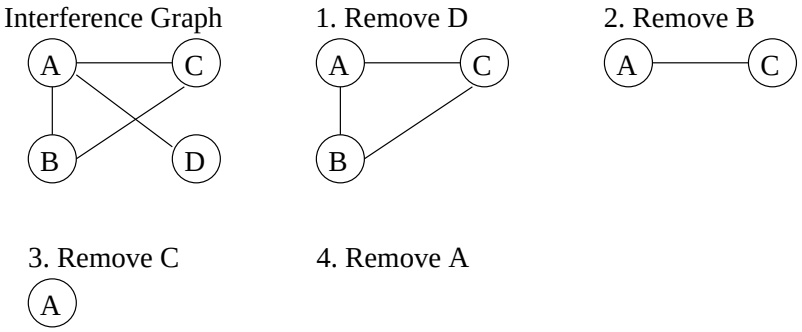

Figure 2.2: Example of removing nodes with degree $<$ k.

can be rebuilt in reverse order of removing the nodes, giving a color to the node as it is reintroduced to the graph.

The approach Chaitin used to find a k-coloring can be shown through the following example. The graph in Figure 2.1 represents an interference graph. For this example, we assume there are only 3 machine registers. Thus, k = 3. Since nodes B, C, and D have degree $<$ k, no matter how their adjacent nodes are colored, there will always be a color available for these nodes. Nodes B, C, and D can then be removed from the graph, along with their adjacent edges. The problem of finding a k-coloring for the graph in Figure 2.1 becomes the problem of finding a k-coloring for the graph now containing only node A. At this point, node A has degree = 0. Thus, node A can also be removed from the graph. Once an empty graph is obtained, the nodes can be colored in reverse order of their removal. These steps of removing nodes and coloring the graph are shown in Figure 2.2 and Figure 2.3.

In the previous example, unconstrained nodes were available to be removed during each

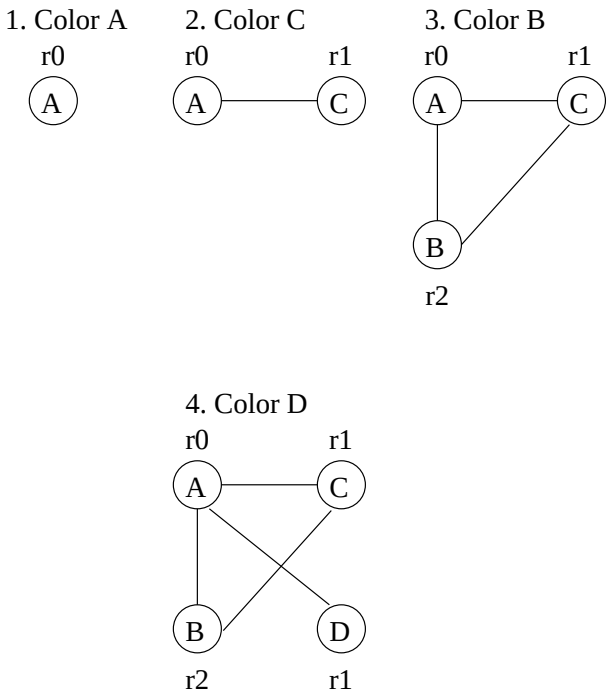

Figure 2.3: Example of coloring the interference graph.

iteration of Chaitin's algorithm. However, this is not always the case. The remaining nodes may all have degrees $\geq$ k. These are known as *constrained* nodes. When the interference graph is left with only constrained nodes, Chaitin's algorithm will choose a node to spill. A store is added after each definition and a load is placed before each use of the spilled variable throughout its live range. This has the effect of splitting the constrained live range into several small live ranges. This will hopefully lower the register pressure at the location in the program where the number of conflicting live ranges was greater than the number of machine registers. After spilling, the graph is rebuilt and another k-coloring is attempted. This process continues until an empty graph and a k-coloring are found.

It is important to understand which particular constrained node is chosen to spill. The goal in choosing a spill candidate is to introduce the least amount of spill code. Since a spill-everywhere approach is used when adding spill code, one heuristic that could be used would be based on the number of uses and definitions of the variable within the live range. However, if the degree of this node is less than the degree of other constrained nodes, removing this node may not unconstrain other nodes, which would result in spilling another

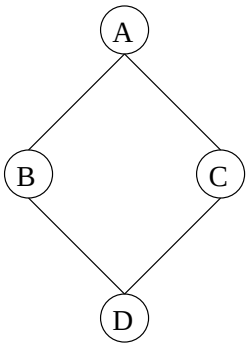

Figure 2.4: Example of 2-colorable graph using Briggs' allocator.

live range. The heuristic Chaitin uses to calculate a spill cost for each node is to divide the execution cost by the degree of the node[16]. For a live range N, this can be written as follows:

$$SpillCost(N) = \frac{Cost(N)}{Degree(N)}$$

$$Cost(N) = \sum_{def \in N} 10^{depth(def)} + \sum_{use \in N} 10^{depth(use)}$$

The depth is the loop nesting level. To spill a contrained node, the node that has the minimum spill cost is chosen. This will be a node with a low execution cost and a high degree.

There is one property of an interference graph that Chaitin's algorithm does not consider. This can be shown by the interference graph in Figure 2.4. If we attempt to find a 2-coloring of this graph, using Chaitin's algorithm, one of the nodes will immediately be chosen to spill. However, a 2-coloring can be found for the graph in Figure 2.4 if nodes A and C, or nodes B and D, are assigned the same color. The work done by Briggs is built upon the fact that constrained nodes may still be colorable if several neighbors of the constrained node share the same color[3, 4, 5].

Briggs' allocator begins by removing unconstrained nodes from the interference graph. If the remaining graph contains only constrained nodes, spilling is delayed. A spill candidate

is chosen and removed from the graph with the hope that, as the graph is later rebuilt, a color will be available for the constrained node. After the constrained node is removed, this will hopefully unconstrain other nodes currently in the graph. This process continues until the graph is empty. The graph is then rebuilt and colored in reverse order of the nodes' removal. At this stage of rebuilding and coloring, if a color is not available for a constrained node, spill code is then added. If spill code was added for a constrained node, the interference graph is rebuilt and another k-coloring is attempted.

Chapter 3

Previous Work

When a constrained node is chosen to spill in a Briggs' allocator, a spill-everwhere approach is used. Live range splitting was introduced as a way to reduce the amount of spill code. The following is a description of previous work on the concept of live range splitting.

3.1 Chow and Hennessy

Live range splitting is used as part of the priority based register allocator implemented by Chow and Hennessy[9]. An interference graph is built in a similar manner to the Briggs' allocator but the coloring algorithm is different. One key difference is when the interference graph is left with constrained nodes after all unconstrained nodes have been removed from the graph. As previously described, Briggs' allocator removes a spill candidate from the graph. If no color can be found for this candidate when the graph is rebuilt, the node is spilled. The priority based allocator splits a live range at this point. This may result in only a part of the live range being spilled or the live range may not need to be spilled at all.

The priority based allocator works with the following steps. Variables live only in a single basic block are first allocated registers. Only global variables, which are live ranges that are live over several basic blocks, become nodes in the interference graph. Each node in the

graph is given a priority based on the cost and savings associated with keeping a variable in a register. The cost is based on the expense of loading a variable from a home memory location to a register and storing it back to the memory location. The savings are a result of reducing the number of memory accesses by keeping the variable in a register throughout its live range.

The conflicts in the interference graph can differ from those in an interference graph built using Briggs' algorithm. Chow and Hennessy define a live range as the set of basic blocks where the variable is live. This can have the effect of adding constraints, through additional edges to the interference graph, between variables where no such conflict would be present in the Briggs' approach. For example, if a variable is live only at the first statement of a basic block and a second variable is live only at the last statement of a basic block, there would be no edge between these variables in the interference graph using Briggs' algorithm. The interference graph, built using Chow and Hennessy's approach, would contain an edge between these variables since they are live within the same basic block. Therefore, the register pressure, or the number of registers required to color the interference graph, may be higher using Chow and Hennessy's priority based allocator.

Once the interference graph is built, register allocation is done within a single pass. All unconstrained nodes are first removed from the graph. If only constrained nodes remain, a priority is computed for each remaining node. The node with the highest priority, which is the live range that would benefit the most from being allocated a register, is assigned a color if one is available. The other nodes are examined to see if they can be split. The splitting is done to allow spilling to occur in only a portion of the live range, if necessary, and to hopefully unconstrain other nodes in the graph.

Chow and Hennessey illustrate their algorithm with the following example. Consider the live ranges for variables A, B, and C, along with the interference graph, in Figure 3.1. These live ranges represent the basic blocks which contain a use or definition of the variable. For this example we assume there are 2 available machine registers. This graph is not 2-colorable. Therefore, each node is given a priority. If B is given a color based on its priority, it is then determined that live range A can be split. The reason for this is A contains basic blocks in the live range that are not adjacent in the control flow graph. Thus, A is known

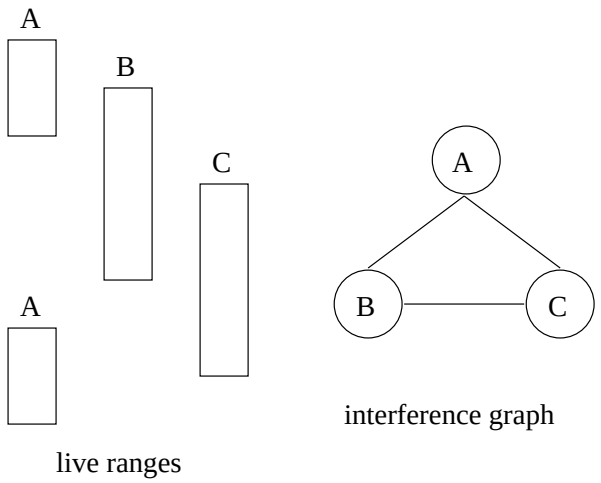

Figure 3.1: Live Ranges and Interference Graph for Priority Based Allocator.

as a *non-contiguous* live range. By splitting A into the live ranges A1 and A2, as shown in Figure 3.2, the interference graph, also shown in Figure 3.2, is now 2-colorable. The Briggs' allocator would have spilled live range A where the priority based allocator has split A and avoided any spilling. The only additional code that is needed is a copy instruction at the entrance block of A2 of the register used in A1 copied to the register used in A2.

We have chosen to implement live range splitting in a Briggs' allocator rather than a priority based allocator for several reasons. One reason is the additional copy instructions that can be added, as shown in the previous example. Depending on the location of these copy instructions, the code could perform worse than the Briggs' spill-everywhere approach. We also feel the finer-grained interference graph built by a Briggs' allocator can color the graph with fewer colors. Although the previous example had the priority based allocator not spilling the live range when a Briggs' allocator would spill, this is not always the case. The priority based allocator will often spill when a Briggs' allocator will not spill. The reason is the priority based allocator reduces the number of available registers by reserving registers for the code generator to produce spill code. Overall, our algorithm builds upon the advantages of the priority based allocator, splitting live ranges so that spill code is not generated for the entire live range, and incorporates those advantages with the advantages of a Briggs' allocator.

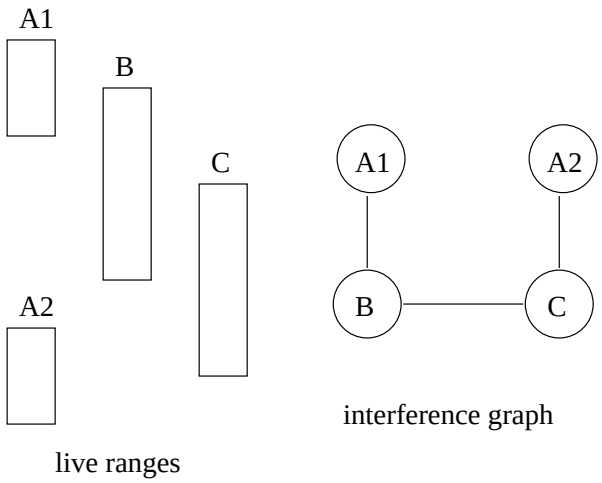

Figure 3.2: Live Range Splitting using Priority Based Allocator.

3.2 Briggs

Briggs proposed aggressive live range splitting to be used in his register allocator[3]. This approach is aggressive because splitting is done to all live ranges that meet certain conditions and splitting is done prior to coloring. Briggs recognizes there are certain locations in the control flow graph where it may be desirable to avoid placing spill code. Moving spill code out of nested loops would be one example. With this criteria, Briggs' algorithm finds these locations in the CFG and all live ranges are split at those locations. It is necessary to add copy instructions at the split locations since splitting essentially divides a single variable into several variables.

According to Briggs, the results obtained using aggressive live range splitting were unsatisfactory. The main reason is that by splitting prior to coloring, an excessive number of splits may be generated with too many copy instructions added. Heuristics were used in an attempt to remove the excess splits but these were not successful. The compile times using aggressive splitting were also a concern. Briggs reported that one algorithm for splitting live ranges around loops increased compile times by as much as 90%. Our live range splitting algorithm avoids these problems since it is a conservative approach. Our live range splitting does not occur until after a spill candidate is chosen. This avoids the problem of excess splits. Our adaptation of Reif and Tarjan's symbolic cover algorithm to find low register

pressure regions keeps compile times from substantially increasing.

3.3 Callahan and Koblenz

Callahan and Koblenz develop a way to reduce the cost of spill code through a concept called register tiling[6]. Their approach is to incorporate the control flow graph structure into the interference graph. They emphasize that the placement of spill code could have either a beneficial or adverse effect on performance. Thus, control flow information would be needed to determine the best locations for spill code. Before an interference graph is built, the basic blocks of the CFG are modeled, as a tree of tiles, where these tiles represent both the control flow and dominator information.

This tree is colored in two phases. There is an initial bottom-up pass where an interference graph is built and colored for each tile. A final top-down pass performs the register assignment. The final phase also decides the location of spill code. For example, it may be possible to move spill code up to a parent tile to improve performance. Register tiling is separate from register assignment. It is feasible that live range splitting could be added to the bottom-up phase where the coloring of each tile takes place.

The work done by Wu[20] points out a disadvantage of the register tiling algorithm. The spill cost function used in the register tiling algorithm does not take into account the degree of the node in the interference graph. Thus, a live range with a small degree may spill that does not unconstrain other nodes in the interference graph. As explained in section 2, this may result in spilling additional live ranges with more spill code being added than would be in the Briggs' allocator.

3.4 Bergner et al.

The recent work by Bergner et al. is similar to our approach[2]. The basic idea of their algorithm is to find regions of high register pressure and then limit spill code to those regions. This is done by first defining *interference regions*. These are the locations in the

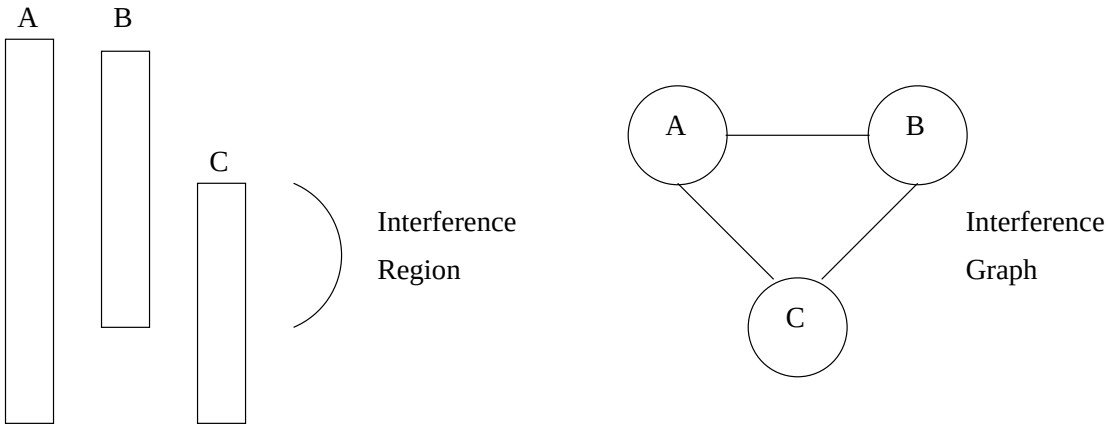

Figure 3.3: Example of a interference region.

CFG where the number of live ranges that are simultaneously live is greater than or equal to the number of physical registers. By removing these regions from two regions that conflict, the edge between the two nodes in the interference graph is removed. This may unconstrain one of the live ranges. Therefore, spill code is only needed in the interference regions.

The interference region spilling algorithm can be demonstrated in the following example. Consider the three live ranges, A, B, and C, and the corresponding interference graph in Figure 3.3. If there are only 2 machine registers available, one of the live ranges must spill. The region where the register pressure is high is also shown in Figure 3.3. If we choose variable C to spill, the interference region is a result of a conflict with C and A and between C and B. To limit spill code to the interference region, one of these edges must be removed from the graph and spill code added in the region where the two live ranges conflict. If we choose to remove the edge between C and B, the resulting interference graph and live ranges are shown in Figure 3.4.

Spilling an interference region is done by adding spill code to the definitions that reach the interference region and to the uses within the region. If a spilled live range extends past the interference region, the variable must be reloaded into memory. If the next use of the split variable, outside the interference region, is within a basic block that is in the interference region, the load is placed in the same location as would occur in a spill everywhere approach. Otherwise, a load of the split variable will be placed at the boundary of the interference

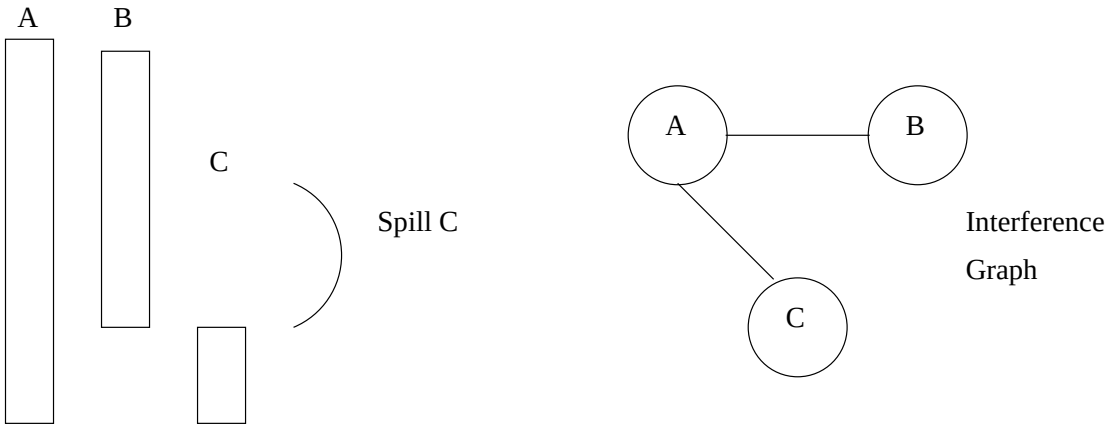

Figure 3.4: Example of spilling an interference region.

region. Since this is an additional load that would not exist in a spill-everywhere approach, it is possible to create slower code than that produced by a spill-everywhere approach. This additional reload instruction could possibly execute more frequently than the loads added by a spill everywhere approach. With profile information, these reloads can be avoided and Bergner's algorithm will choose to spill using the Chaitin heuristic.

Our algorithm is similar to the Bergner et al. approach, but different in implementation. The Bergner approach identifies high register pressure regions, known as interference regions. Our algorithm finds areas of low register pressure. It is important to note that the Bergner definition of interference and noninterference regions is more fine-grained than our corresponding definitions of low and high register pressure regions. Our algorithm, as will be discussed, places boundaries at the basic block level. Bergner's algorithm identifies boundaries between high and low register pressure areas within a basic block.

The work done by Bergner points out that the spill-everywhere approach is overly pessimistic. In the C benchmarks tested by Bergner, there was an average 33.6% reduction in the number of dynamic spills and a 8.3% reduction in the execution time of those benchmarks. The downside to these improvements was the cost of compile times. Bergner reported that their algorithm increased compile times by 20 - 40%. Even though our algorithm is more coarse grained in its definition of low register pressure regions, we hoped to obtain similar reductions in the number of dynamic spills and execution times. We also felt we

could accomplish these results without paying a substantial penalty in compile times. The reason for this is our algorithm uses an adaption of Reif and Tarjan's efficient algorithm for finding symbolic covers to compute the regions of low register pressure, which will be shown to be almost linear in its complexity[17].

One aspect of an interference region was not examined by Bergner. It was previously mentioned that the register allocator chooses a spill candidate based on the minimum cost that is calculated for each live range. One part of this cost included the execution frequency of definitions and uses of the variable within its live range. However, if a number of uses and definitions of a live range are outside an interference region, which is a low register pressure region, it may be beneficial to ignore these references in the frequency calculation. Since a minimal amount of spill code would potentially be added outside an interference region, it may be better to spill this node even if its execution frequency is high. We decided to extend our algorithm of live range splitting to examine whether using a spill cost heuristic, which calculated execution frequencies based only on references within high register pressure areas, would have a beneficial effect on the spill choices made by the register allocator.

Chapter 4

Live Range Splitting

4.1 Introduction

The basic idea of our algorithm is to identify regions in the control flow graph with low register pressure. In the basic blocks of these regions the number of registers required will be less than or equal to the number of available registers. In a typical Briggs' register allocator, once a variable is chosen to spill, the variable will spill everywhere. With our algorithm, the number of loads and stores added for a spilled variable can be reduced. If a variable spills in a region of low pressure, only a load will be needed at the entrance to the region. If there is a definition within the region of low pressure, a store is only needed at the exit point of the region.

This can be illustrated by the following example. In Figure 4.1, Example A, three variables and their live ranges are represented. If we limit the number of machine registers to one for this example, it will be necessary for one of the variables to spill. If variable A spills, a load will be placed before each use and a store placed after each definition throughout the entire live range. However, there is a region where variable A has no conflicts. This area of low register pressure is shown in Figure 4.1, Example B. By identifying this region, only a load of A is needed at the entrance to the region along with a store of A at the exit of the region. This is illustrated in Figure 4.1, Example C.

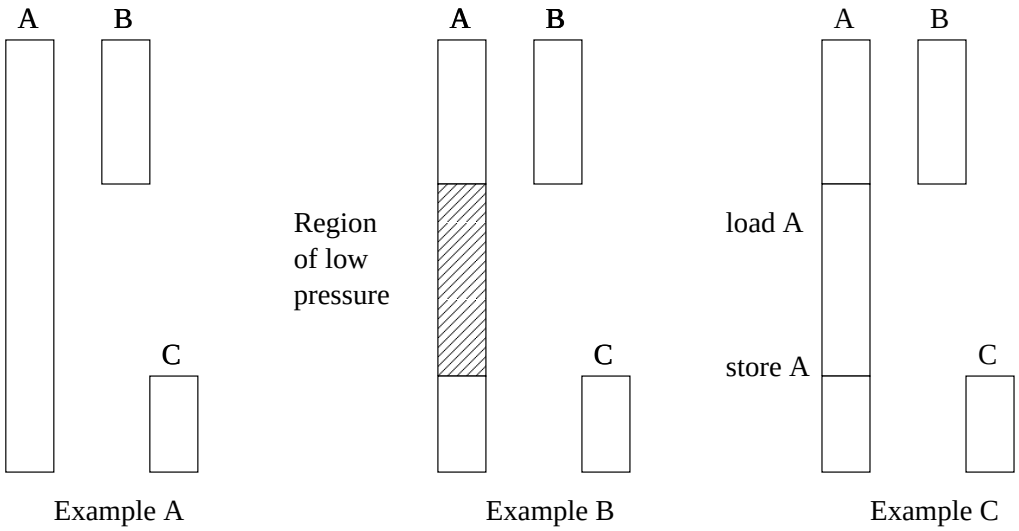

Figure 4.1: Example of a low register pressure region.

One advantage of our algorithm is that it is incorporated into the stages of a Briggs' register allocator. The stages of a Briggs' allocator are shown in Figure 4.2. The step of computing regions of low register pressure can be done prior to computing spill costs, if region information is used in the spill cost heuristic. Since the spill cost of a live range may be modified if there are references in low register pressure regions, it is necessary to first compute regions of low register pressure. This will add to the register allocation execution time since low register pressure regions are found even if no live ranges will end up spilling. If the normal spill cost heuristic is used, low register pressure regions are computed prior to adding spill code. This minimizes the increase in register allocation time since low register pressure regions are found only if necessary. Adding spill code at the boundaries of a low pressure region is done prior to adding spill code. Our modified register allocator is shown in Figure 4.3.

Another advantage to live range splitting is our algorithm's efficiency. Finding a region of low register pressure involves examining the register pressure of individual basic blocks in the CFG. As mentioned previously, this is done through a modification of Reif and Tarjan's algorithm for computing symbolic covers. Since this algorithm is almost linear in its running time, live range splitting is done without a large increase in compile time.

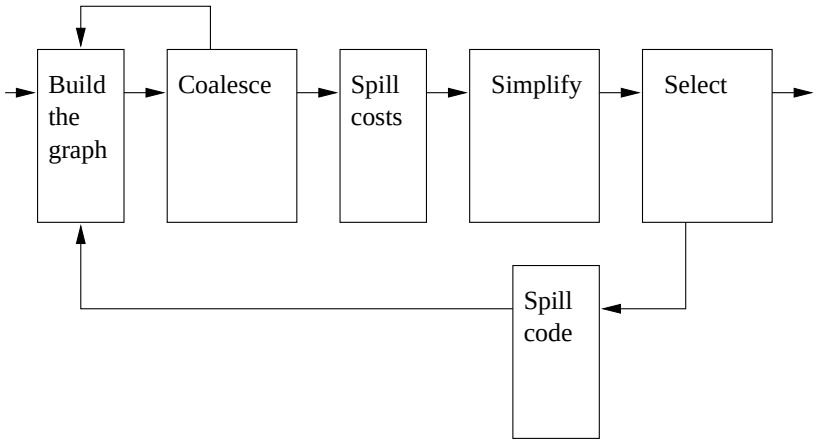

Figure 4.2: Briggs' Register Allocator

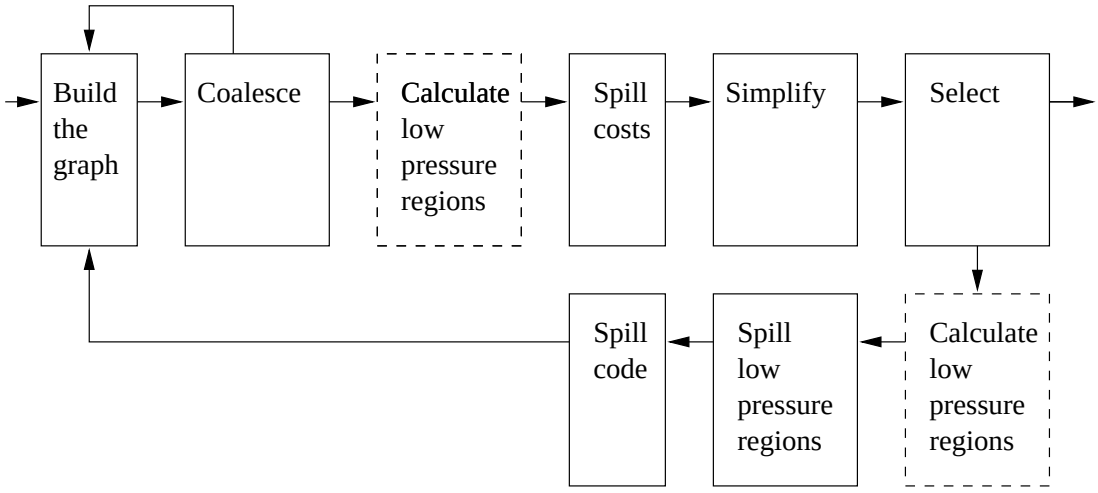

Figure 4.3: Modified Briggs' Register Allocator Using Live Range Splitting

4.2 Low Register Pressure Regions

The first step in our algorithm is computing regions of low register pressure. Initially, each basic block is considered before larger regions of low pressure are calculated. These larger regions are computed efficiently using dominator information and modifying Reif and Tarjan's algorithm for computing symbolic covers.

The dominator information is obtained from the dominator tree generated from the CFG. A dominator block A of basic block B is a block where every path from the start node in the CFG to B must pass through A. If block A is dominated by every other dominator of block B, block A is the parent of block B in the dominator tree. Block A is also known as the *immediate dominator* of block B. This means there are no dominators of B on any path leaving A to B. Similar definitions exist if the CFG is examined with its edges reversed. A post-dominator block A of basic block B is a block where every path from the exit node in the CFG to B must pass through A. The definition for a post-immediate dominator is parallel to the definition for immediate dominator. The post-dominator information is used when spill code for definitions within a low register pressure region are added.

In order to compute regions of low register pressure, we need to determine the register pressure for each basic block and the maximum register pressure between a block and its immediate dominator. We define **pressure** of a basic block as the maximum number of variables live and available at any statement in the block. In order to accurately calculate the pressure at each basic block it is also necessary to look at the variables that are live-out of each predecessor block of the current block. The reason is that a variable may be considered dead at a particular statement and therefore, would normally not conflict with any variables defined at that statement. However, if the dead variable and a defined variable are live-out of a predecessor block, a conflict exists between those variables in the interference graph. Therefore, this dead variable is included in the live set for the statement when register pressure is being calculated.

We define **ipressure** of a basic block as the maximum pressure of any block on any path between the block and its immediate dominator. The ipressure information is used to compute regions of low register pressure containing several basic blocks. For example,

consider a block B with an ipressure less than the number of available machine registers. If a variable spills in the basic blocks between B and its immediate dominator, the amount of spill code can be reduced. A load in the immediate dominator of B will dominate each use of the spilled variable. Spill code for each use will not be needed in the basic blocks between B and its immediate dominator since no block in that region has a pressure greater than the number of machine registers.

The ipressure of a basic block is efficiently calculated since this computation is based on the algorithm given by Reif and Tarjan for computing symbolic covers, or the mapping of symbolic expressions to expressions in the program. This mapping is typically used for different compiler optimizations. In order to find symbolic covers, it is first necessary to compute an *idef* set for each basic block. It is this part of the symbolic cover algorithm that we adapt to finding low register pressure regions. The **idef** of a basic block B is the number of definitions in the blocks on all paths between B and its immediate dominator[17]. Our definition of ipressure is analogous to idef. The Reif and Tarjan algorithm is adapted to find the maximum pressure values in the blocks on all paths between a basic block and its immediate dominator.

With both pressure and ipressure calculated for every basic block in the CFG, regions of low register pressure are then computed. This is done with the following steps:

1. Basic blocks are examined in reverse depth-first search order. Therefore, each block is examined before its dominators.

2. The basic block is marked as visited.

3. If the basic block's pressure is less than or equal to the number of available registers, the block is added to the current region of low pressure. Otherwise, a new region is started if the current region is not empty.

4. If the basic block's ipressure is less than or equal to the number of available registers, every block on a path from the block's immediate dominator to the block is marked as visited and added to the current region. Otherwise, a new region is started if the current region is not empty.

```
RegionAnalysis (Block B, Region C, BlockList L, RegionList R)
  B.Visited ← true
  if (B.pressure ≤ #regs)
    C.Add ← B
  if (B.ipressure ≤ #regs)
    for each Block, W, between idom(B) and B
      W.Visited ← true
      C.Add ← W
  if ((B.pressure ≤ #regs OR B.ipressure ≤ #regs)
       AND (B != StartNode))
    RegionAnalysis(idom(B), C, L, R)
  else
    if (C.IsNotEmpty)
      R.AddRegion ← C
      C ← ∅
    if (L.IsNotEmpty)
      RegionAnalysis(L.NextUnvisited, C, L, R)
```

Figure 4.4: Algorithm for computing low register pressure regions.

5. If the condition in either step 3 or step 4 is true, step 2 is repeated with the immediate dominator of the basic block. Otherwise, step 2 is repeated with the lowest unvisited basic block.

These steps are also presented in Figure 4.4.

The efficiency of computing low register pressure regions is shown in the following analysis. First, consider the size of the CFG. Let m be the number of edges in the graph and n be the number of nodes. Computing the pressure of each basic block is in O(n) since each basic block is examined once. Computing the idef of each block, or the ipressure, can be done in O(mα(m,n)), where α(m,n) is a functional inverse of Ackermann's function[15, 17]. This computation is essentially linear, since α(m,n) $\leq$ 4 for all practical purposes[10].

4.3 Computing Load and Store Candidate Blocks

Once regions of low register pressure are computed, the algorithm determines where loads and stores for a spilled variable in the region can be safely placed. The location of a load for a spilled variable within a region of low pressure must be in a basic block that dominates all the uses of the variable in that region. The location of a store must be in basic block that post-dominates all the definitions in the region. The location for load candidate blocks is looked at first. An analogous algorithm is used to find store candidate blocks.

The construction of low register pressure regions indicates where the load candidate blocks can be found. Several blocks on all paths between block B and its immediate dominator are added to a region if B's ipressure is less than or equal to the number of machine registers. Therefore, a load could be placed in the immediate dominator of B for each use of a spilled variable within the low pressure region. There may be several blocks within the region that are possible candidates for a load. It may also be possible that a block outside the region will be the dominator of uses in the region. In order to find load candidate blocks, it is necessary to first find the **iuse** set for each basic block. The iuse set is the number of uses of a variable in the blocks on all paths between a basic block and its immediate dominator[18].

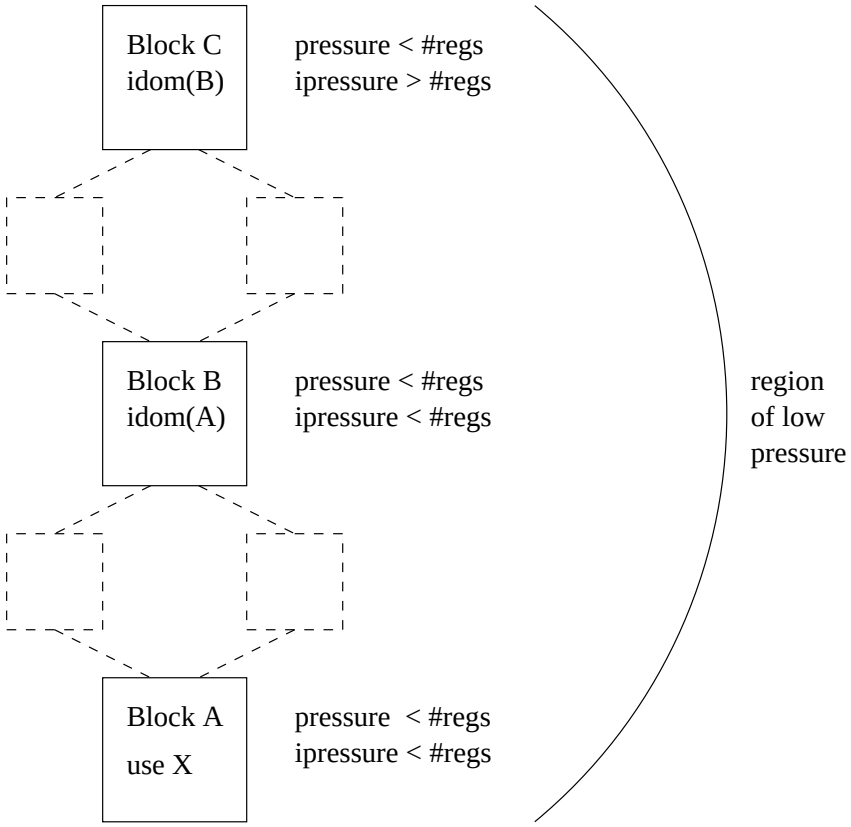

Figure 4.5: Low register pressure region.

The previous concept can be illustrated as follows. Consider a block A, such that the block's pressure and ipressure are less than the number of machine registers. This condition will also hold for the immediate dominator of A, block B. Finally, the immediate dominator of B, block C, has a pressure less than the number of machine register, but C's ipressure is greater than the number of registers. This is shown in Figure 4.5.

Let variable X be a spilled variable with a use in block A. If X is not contained in A's iuse set and there are no uses of X in idom(A), block B, a load of X can be placed in A or block B. If X is not contained in B's iuse set and there are no uses of X in the idom(B), block C, a load of X can be placed in C as well. The following is a description of how these candidate blocks are computed.

1. For each spill variable X, each low pressure region is examined separately. The block

B that is deepest in the dominator tree is initially considered.

2. If the spilled variable X is in the use set of B, the list of load candidate blocks is cleared. B is then added to the list of candidate blocks.

3. If the spilled variable X is in the iuse set of B, then the list of load candidate blocks is cleared.

4. If the iuse set of B does not contain X, then the immediate dominator of B is added to the list of candidate blocks.

5. Step 2 is repeated with the parent of B in the dominator tree.

The algorithm continues until the dominator block of the entire region is reached. This block may or may not be in the region. First, consider the case where the block is in the region. If X is in the use set of the block, the candidates set is cleared and this block becomes the only load candidate block. If X is not in the use set, this block is added to the candidate set.

It is possible that the only dominator block of a low pressure region may not be a member of the region. The only case to consider is if the candidate set is empty. In order to consider this block as a possible load candidate, the register pressure at the exit point of the block has to be calculated. If this register pressure is less than the number of available registers, the load can be placed at the end of the block. Otherwise, the load candidate will become the set containing each entrance block to the low register pressure region. The basic algorithm for finding load candidates is presented in Figure 4.6.

There is one special case to this algorithm. Consider the low pressure region in Figure 4.7. If X is a spilled variable, it would be desirable to load X upon entrance to the low register pressure region. However, a load for X cannot be placed in block A since this will reload X upon each iteration of the loop. In this situation, if block A is chosen as the load candidate, a new block is created upon entrance to A, with A's predecessors not in the loop becoming predecessors of the new block, and the load is added to this new block.

A similar algorithm is used for computing store candidates. The idea is to find blocks that

```
FindCandidate (Block B, CandidateSet S, Region R, Variable X)
  if (B.IsRegionDominator)
    if (B ∈ R)
      if (X ∈ B.use)
        S ← ⊘
      S.AddCandidate ← B
    else if (S.IsEmpty)
      S.AddCandidate ← B
  else
    if (X ∈ B.use)
      S ← ⊘
      S.AddCandidate ← B
    if (X ∈ B.iuse)
      S ← ⊘
    if (X ∉ B.iuse)
      S.AddCandidate ← idom(B)
    FindCandidate (idom(B), S, R, X)
```

Figure 4.6: Algorithm for finding load candidates.

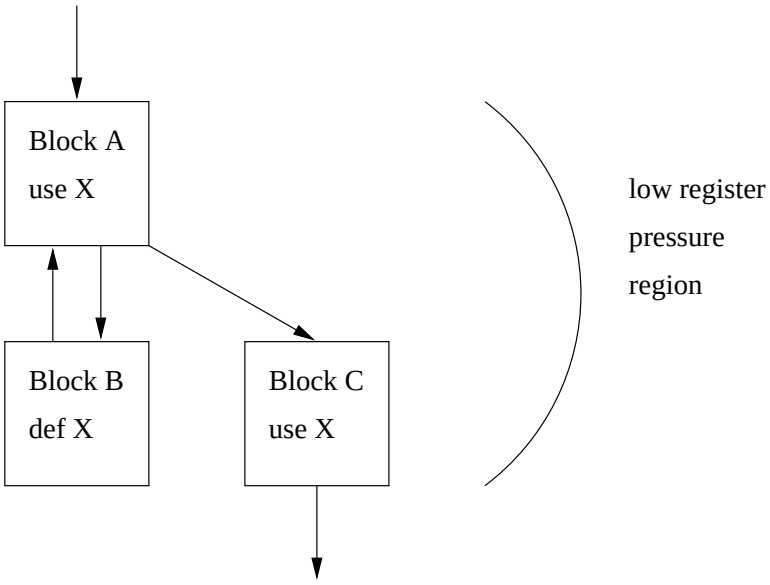

Figure 4.7: Special Case to Computing Load Candidates.

post-dominate each definition in the region. The steps for finding store candidates are as follows.

1. For each spill variable X, each low pressure region is examined separately. The block B that is deepest in the post-dominator tree is initially considered.
2. If the spilled variable X is in the def set of B, then the list of store candidate blocks is cleared. B is then added to the list of candidate blocks.
3. If the spilled variable X is in the idef set of B, then the list of store candidate blocks is cleared.
4. If the idef set of B does not contain X, then the post- immediate dominator of B is added to the list of candidate blocks.
5. Step 2 is repeated with the parent of B in the post-dominator tree.

It is possible that there may not be a post-dominator block of each definition that is a member of the region. In this case, the store candidate then becomes the set containing each exit block of the region.

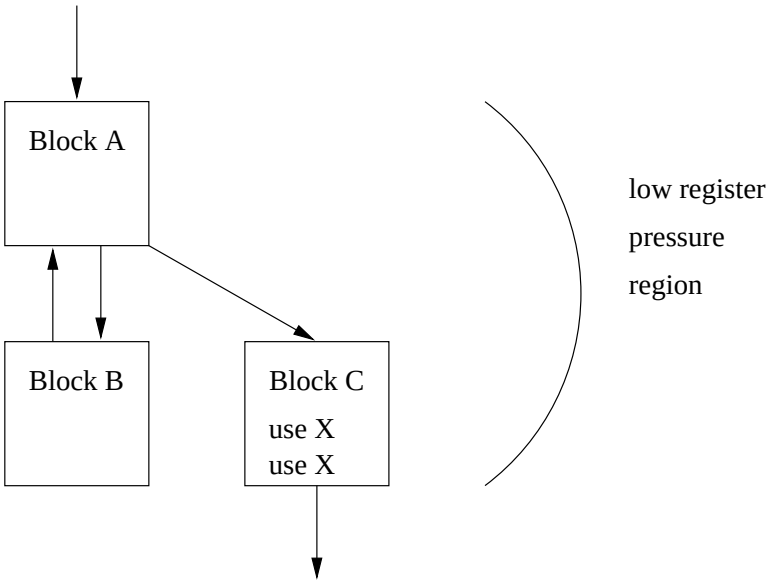

Figure 4.8: Choosing a Load Candidate.

The algorithm for computing load and store candidate blocks is efficient. We need to consider the number of spilled variables and the number of basic blocks in the CFG to analyze the running time. Let p be the number of spilled variables and n be the number of basic blocks in the CFG. For each spilled variable, each basic block would have to examined in the worst case. Therefore, computing load candidate blocks is in O(pn).

4.3.1 Choosing A Load or Store Candidate

The goal for this step is to find the cheapest location to insert spill code in the low register pressure region. There may be several load and store candidate blocks that have been computed from the previous step. It is also possible that a load candidate block or store candidate block can be more expensive, in terms of execution frequency, than the execution cost for normal Chaitin spilling. For example, consider the low pressure region in Figure 4.8. If X spills, block A will be a load candidate block. If a load is placed in this block, depending on how often the loop iterates, there could be considerably more loads executed than would be executed if normal Chaitin spilling is used.

In order to determine which candidate block to load or store the spilled variable to a register for the low pressure region, profile information is collected for the execution frequency of each basic block in the CFG. With this information, the cheapest load and store candidate block is found. This execution frequency of these candidate blocks is compared to the cost of performing normal Chaitin spilling. If it is determined that inserting spill code in the candidate blocks is costlier than normal spilling, our algorithm reverts to a spill-everywhere heuristic. Using this approach, live range splitting will perform at least as well as normal Chaitin spilling if not better.

If it is determined that live range splitting is beneficial, splitting is done in the following manner. The spill code is added through the addition of a **load** $\mathbf{X}'$ in the load candidate block or a **store** $\mathbf{X}'$ in the store candidate block. Each use or definition of $\mathbf{X}$ in the region is replaced with the variable $\mathbf{X}'$.

4.4 Using Low Pressure Regions in Spill Cost Heuristics

As was previously discussed, spill cost heuristics are partially based on the execution frequency of a variable within its live range. This heuristic can be modified to count only those uses and definitions of a variable not contained in a low register pressure region. The cost function is modified as follows, where N is each live range in the interference graph and LPR stands for low pressure region.

$$Cost(N) = \sum_{def \in N \wedge def \notin LPR} 10^{depth(def)} + \sum_{use \in N \wedge use \notin LPR} 10^{depth(use)}$$

The reasoning for this function is to develop a heuristic that will choose a variable with most of its references in low register pressure regions. This is an attempt to take of advantage of live range splitting since the only spill code needed for those references will be at the entrance and exit points of the low register pressure region. This heuristic can allow more expensive live ranges, in terms of execution frequency, to spill if the amount of the spill can be reduced through live range splitting.

```
L1:  A <- function()
     B <- A * 3
     C <- A
     if B = 0 goto L3
L2:  C <- C + 3
     B <- B - 1
     if B > 0 goto L2
L3:  D <- A + C
     if A > 0 goto L6
L4:  if A = 0 got L7
L5:  A <- A + D
     goto L7
L6:  A <- A - D
L7:  E <- A + D
```

Figure 4.9: Code for algorithm example.

4.5 Example

The advantages of the algorithm can be demonstrated in the following example. Figure 4.9 illustrates a sample code segment that will be used. Figure 4.10 illustrates the CFG of this example and the pressure and ipressure values for each basic block. If we limit the number of machine registers to 2, this example will not be 2-colorable. If we choose to spill the variable A, we can limit the amount of spill code added to the code by finding regions of low register pressure. One region exists in this code containing blocks 7, 5, 6, 4, 3. Figure 4.11 shows A spilled everywhere in a typical Briggs' allocator. Figure 4.12 shows A spilled in regions of high pressure and at the boundary of the low register pressure region. Our algorithm is able to reduce the number of loads from 8 to 3 and the number of stores from 3 to 1.

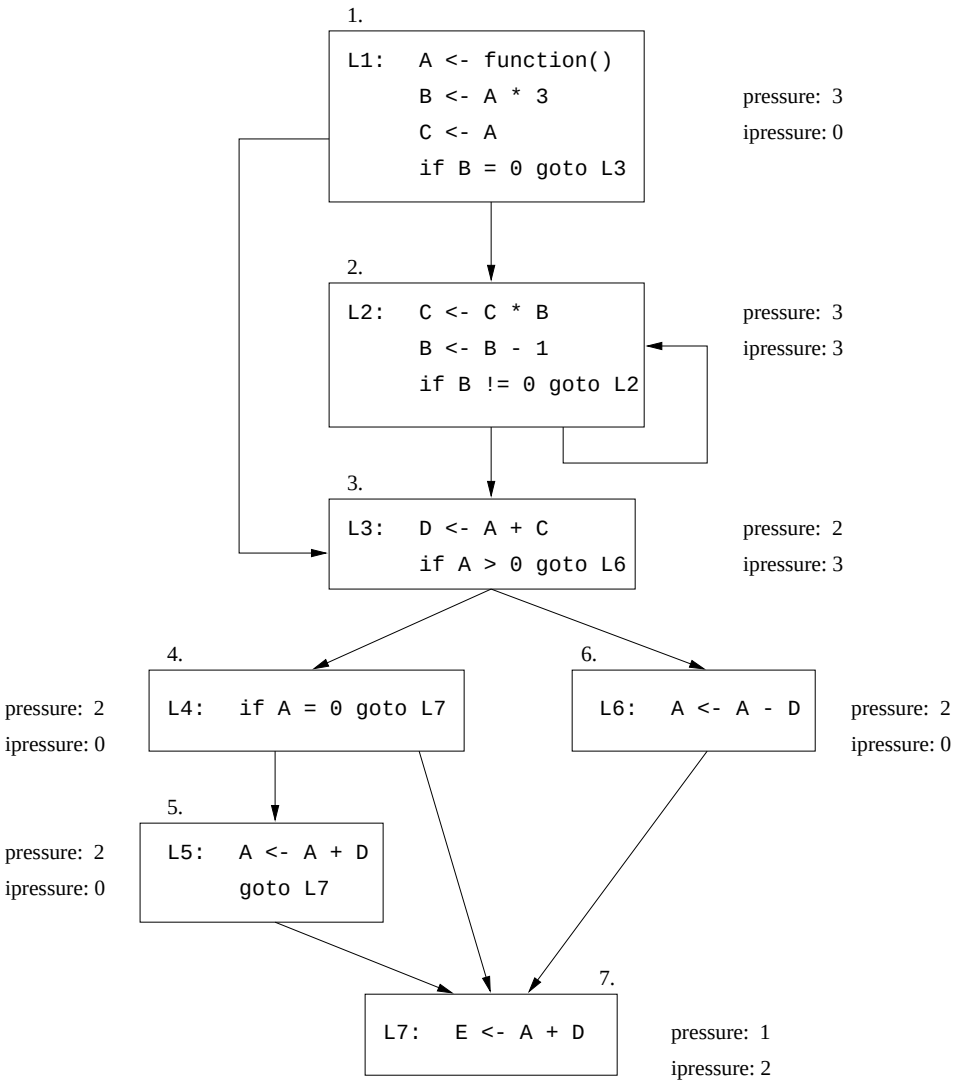

Figure 4.10: Example CFG with pressure and ipressure values.

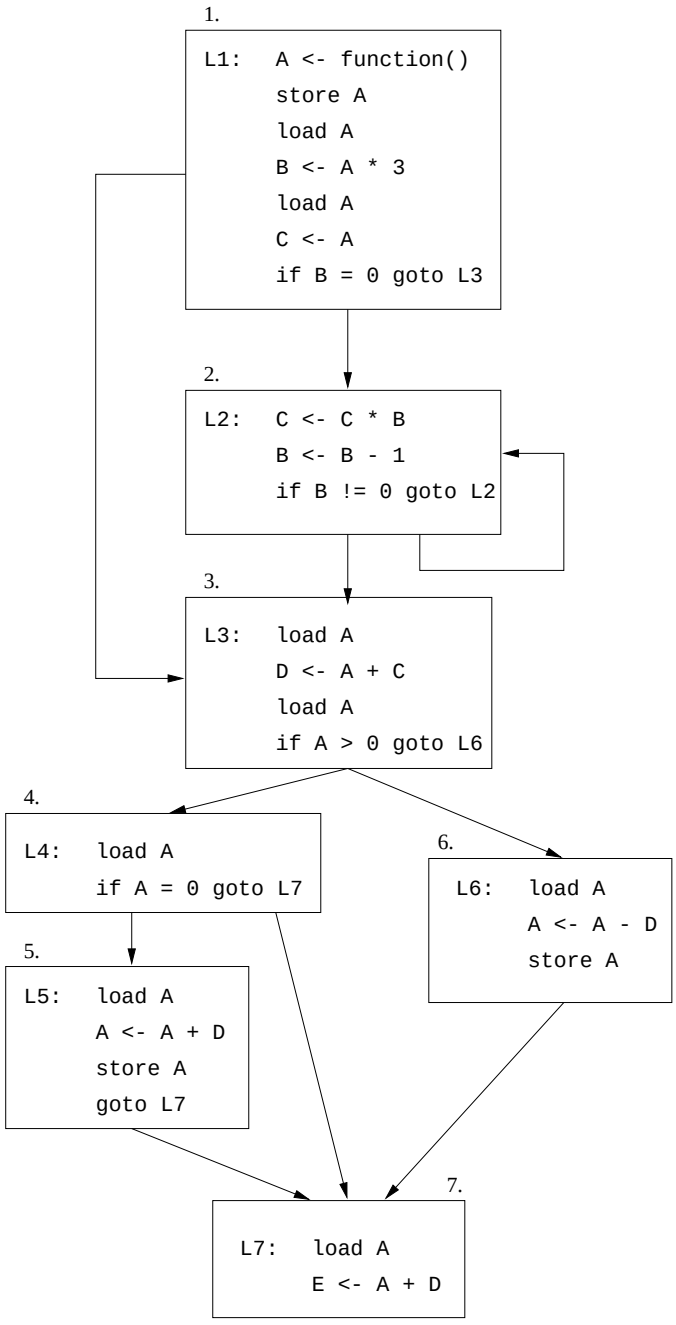

Figure 4.11: Variable A spilled everywhere.

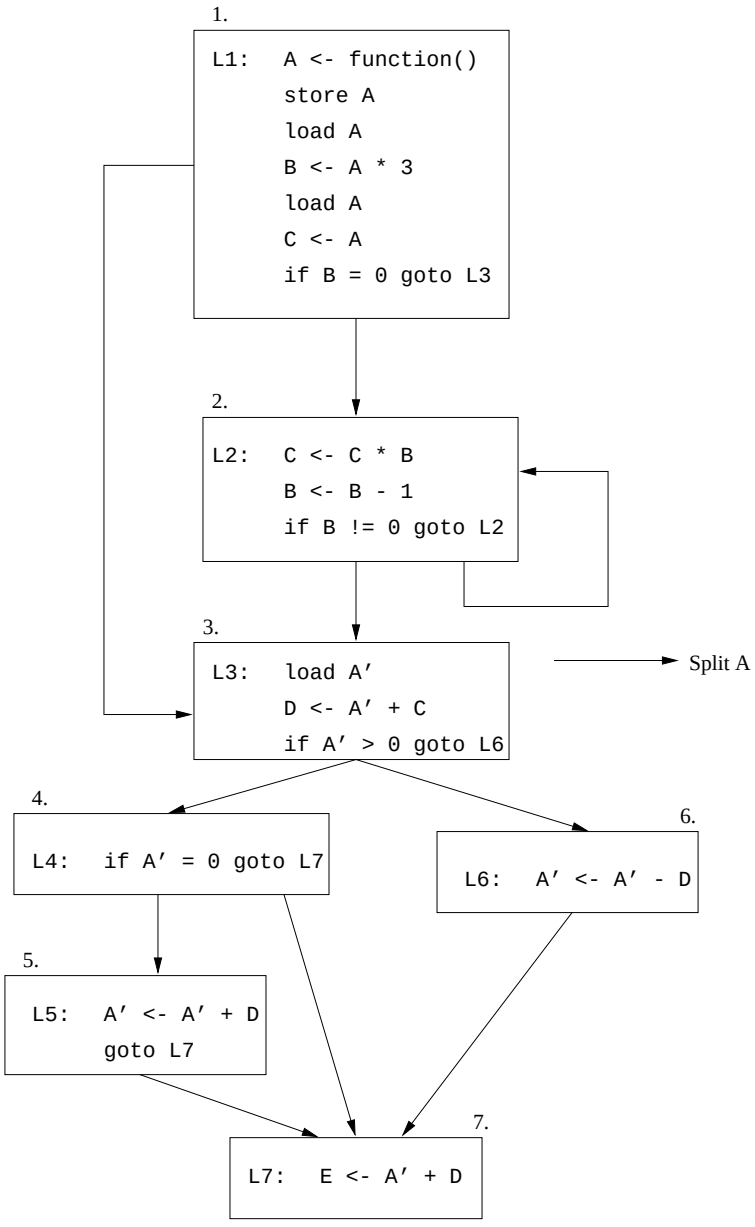

Figure 4.12: Variable A spilled with low register pressure regions identified.

Chapter 5

Experimental Results and Analysis

The live range splitting algorithm has been implemented in the Rocket compiler. Rocket is an optimizing compiler that is designed to be retargetable for a number of ILP architectures[19]. Rocket performs register allocation using Briggs' graph coloring algorithm. Live range splitting has been incorporated into Rocket's register allocator through the addition of the following steps.

1. Live range splitting is performed during early register allocation, prior to instruction scheduling.

2. If low register pressure region information is not used in the spill cost heuristics, register pressure and ipressure values of each basic block, along with low register pressure regions, are computed once the register allocator determines it is necessary to spill a live range.

3. Prior to adding spill code, profile information is used to determine if it is beneficial to split a live range in a low register pressure region. Load instructions are added to the load candidate blocks and store instructions are added to the store candidate blocks at this point.

4. If low register pressure region information is to be used in the spill cost heuristics, register pressure and ipressure values are calculated for each basic block prior to building

the interference graph. The register pressure values are used when determining the spill cost for each live range in the interference graph.

5. Live range splitting is used to generate assembly code for the Alpha 21164 processor[13]. The Alpha 21164 is a superscalar architecture, with 2 integer and 2 floating point pipelines, that can issue up to 4 instructions per cycle.

5.1 Benchmarks

The following C benchmarks were tested on the Alpha.

1. *my8q*: A program that solves the eight queens problem that has been extended to solve the problem 20,000 times.
2. *bigBubble*: A bubble sort program that sorts an array of 40,000 elements.
3. *gauss*: A program that uses Gaussian elimination.
4. *nsieve*: A program that implements the Sieve of Eratosthenes.
5. *matrix_mult*: A program that multiplies 2 512x512 matrices.
6. *myWhetstone*: A program that tests compiler optimizations and the performance of floating point operations[11].
7. *killcache*: A program designed to test the performance of data caches.
8. *livermore*: A program to evaluate the execution time of 14 loops.

5.2 Experimental Setup

In order to evaluate the effectiveness of our live range splitting algorithm, we needed to determine if the number of memory references executed by a program was reduced. This would hopefully lower the execution time of the program. We also wanted to determine if

these benefits were gained without substantially increasing the compile time of a program. To perform this analysis, the following data was collected for each benchmark.

1. The number of static spills generated by the register allocator.
2. The time, in seconds, it takes the compiler to complete register allocation.
3. The number of dynamic load and store instructions executed by the program.
4. The execution time, in seconds, of the program.
5. The average number of basic blocks in a low register pressure region.

The last measurement was collected in order to determine how many basic blocks are in a typical low register pressure region. This is to see if the low register pressure regions are large enough to take advantage of the live range splitting algorithm.

For each benchmark, the number of available machine registers was limited in order to cause the benchmark to spill. Table A.1 shows each benchmark and the limit set for available integer and floating point registers. For example, one test with the *my8q* benchmark was done by limiting the number of integer registers to 9 and the number of floating point registers to 9. A second test of the *my8q* benchmark was done by limiting the number of registers to 8.

5.3 Results

5.3.1 Using Chaitin's Spill Cost Heuristics

This section graphs the data collected for each benchmark when using a spill-everywhere approach and when live range splitting was performed. For this first set of results, the normal Chaitin spill cost heuristic was used. When a spill variable needs to be removed from the interference graph, the choice is based on a minimum spill cost as described in Chapter 2. These graphs show the normalized results for each test. The results for normal

Benchmark	Integer Registers	Floating Point Registers
my8q	9	9
my8q	8	8
bigBubble	10	10
bigBubble	8	8
gauss	10	10
gauss	8	8
nsieve	20	20
matrix_mult	9	9
matrix_mult	8	8
myWhetstone	9	9
myWhetstone	8	8
killcache	9	8
killcache	8	8
livermore	20	20
livermore	12	12

Table 5.1: The number of available registers for each benchmark.

spilling, or the spill-everywhere approach, is set to 100% and the results for live range splitting are a percentage of the results obtained using normal spilling. The numbers that were used to generate these graphs can be found in the Appendix at the end of this thesis.

The Rocket compiler was modified to output the number of loads and stores that were inserted for normal spilling and the number of loads and stores inserted when live range splitting was performed. The graphs in Figure 5.1 shows the normalized results for each benchmark. For example, the *my8q* benchmark, with 9 integer registers and 9 floating point registers, reduced the number of static spills by 60% over the normal spill everywhere approach.

Live range splitting was effective in reducing the number of load and store instructions added by Rocket's register allocator. The largest reduction in static spills was seen in the *my8q* and *bigBubble* benchmarks. Considering only these test cases, there was an average 38.97% reduction in the number of static spills added to the program. The average for all the benchmarks was a 14.87% reduction in static spills. Live range splitting was not

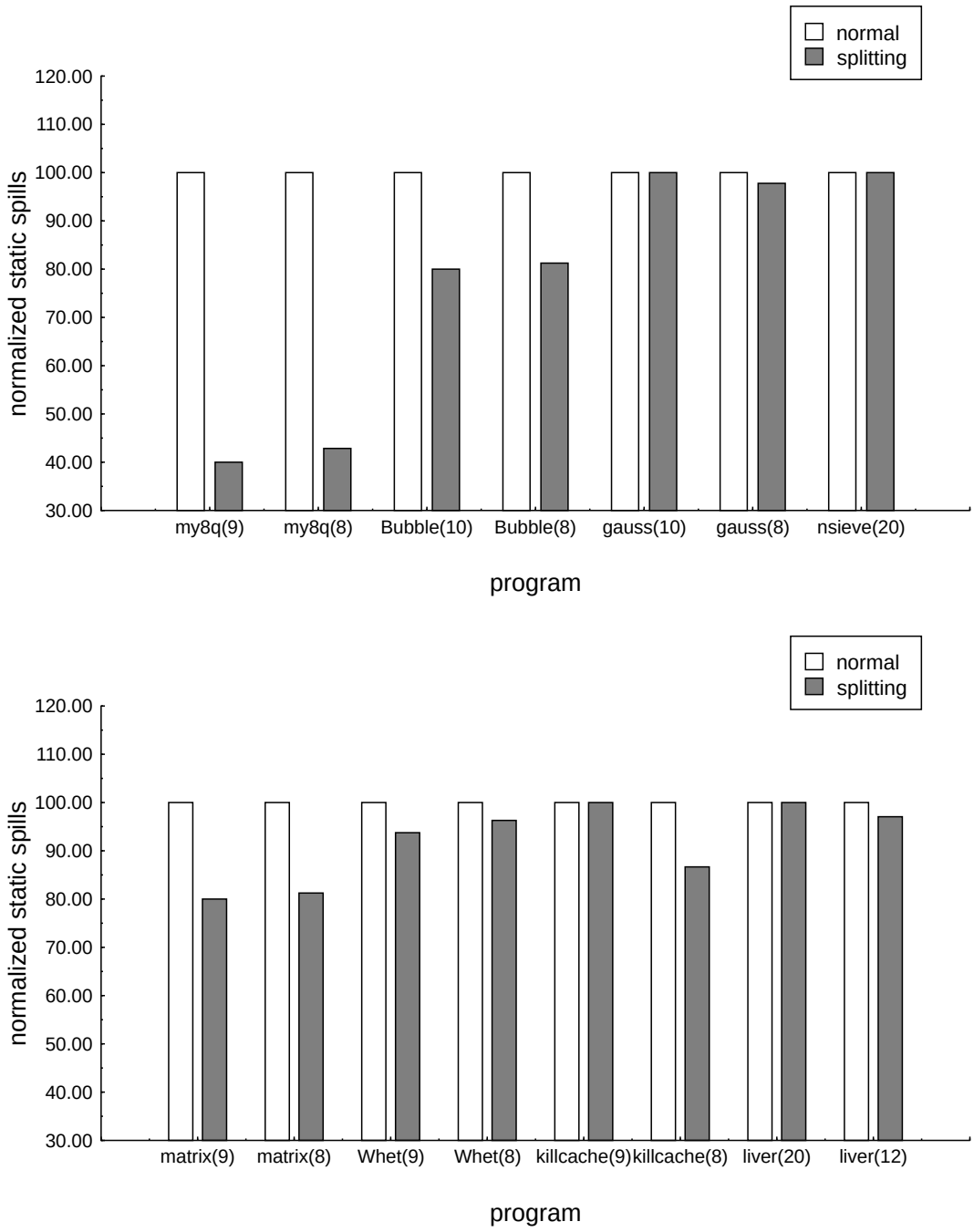

Figure 5.1: Reduction in static spills using normal spill cost heuristic.

effective in every case. The benchmarks, *gauss* with 10 registers, *nsieve*, *killcache* with 9 registers, and *livermore* with 20 registers, generated the same number of spills as the normal spilling approach.

The number of dynamic load and store instructions executed when using normal spilling and live range splitting was counted through the use of Digital Equipment's profile tool ATOM[12]. As shown in Table A.2, live range splitting did not reduce the number of dynamic store instructions. The reason for this is that there are not enough definitions of a spilled variable to take advantage of live range splitting. The majority of spilled variables contain only 1 or 2 definitions in the live range. As will later be seen, the average size of a low register pressure region may not have been large enough to move the store instruction to a more beneficial store candidate block. Therefore, the graphs in Figure 5.2 show the normalized results for the number of dynamic load instructions for each benchmark. For example, the benchmark *my8q*, with 9 registers, was able to reduce the dynamic loads by 16.6%.

The benchmarks that reduced the largest number of static spills reduced the most dynamic load instructions. The *my8q* and *bigBubble* programs reduced the number of dynamic loads by an average 18.34%. The other benchmarks had either no reduction in the number of loads or the amount was negligible. For example, the *gauss* benchmark, with 8 registers, had a reduction of only 512 loads. The *livermore* benchmark, with 12 registers, executed 372,500 more load instructions using live range splitting. The average reduction in dynamic loads for all the benchmarks was 4.89%.

The execution time of each benchmark was measured by recording user time for the program. The execution times shown in Table A.3 are an average of ten runs of the benchmark program. The graphs in Figure 5.3 represent the normalized results for each benchmark. For example, the *my8q* benchmark, with 9 registers, improved execution time by 4.81%.

The largest improvement in execution time was seen in the same benchmarks that saw the largest improvement in static spills and dynamic loads. The benchmarks, *my8q* and *bigBubble*, reduced execution times by an average 10.58%. There was little difference in execution times for the other programs. This should be expected since there was no change

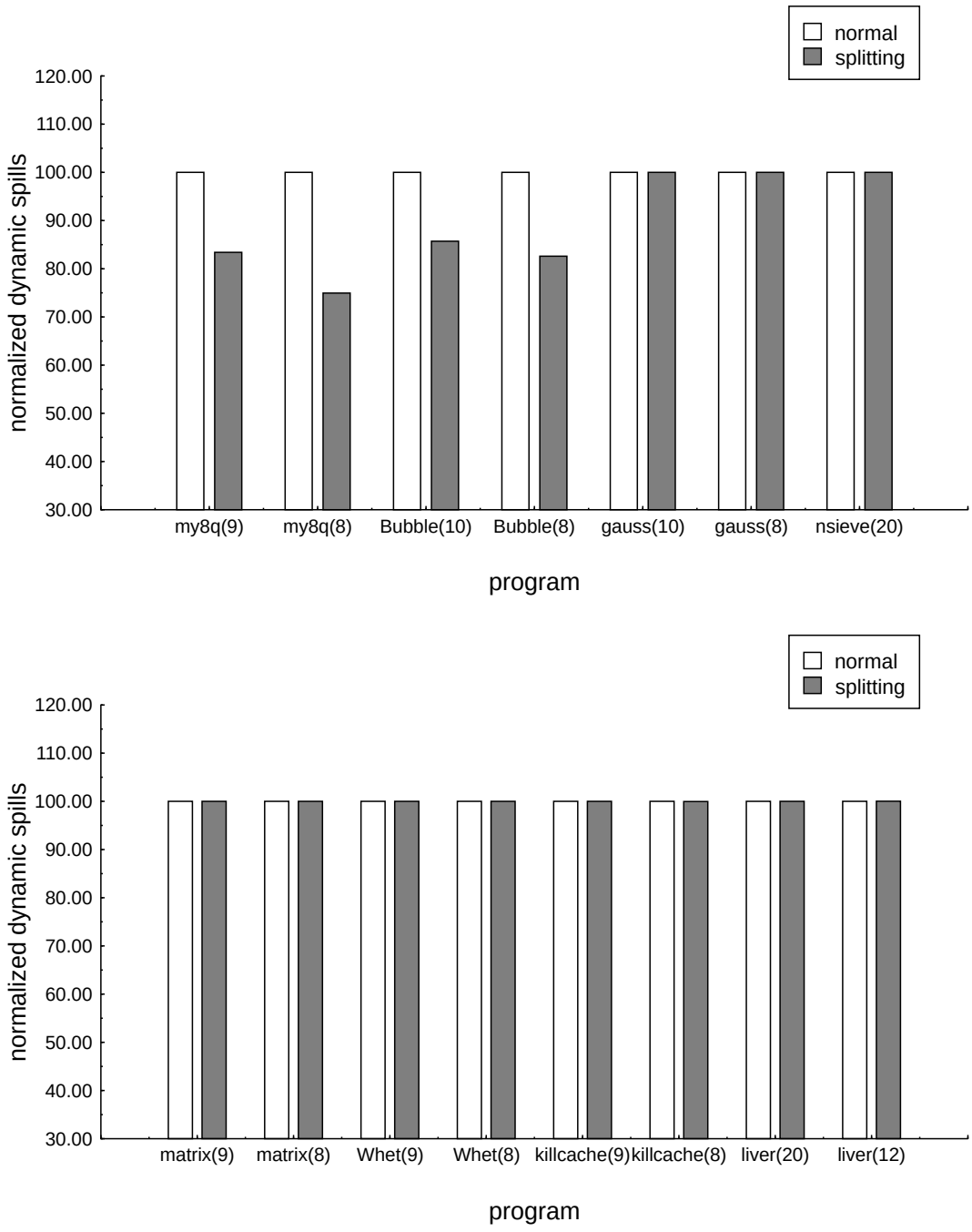

Figure 5.2: Reduction in dynamic loads using normal spill cost heuristic.

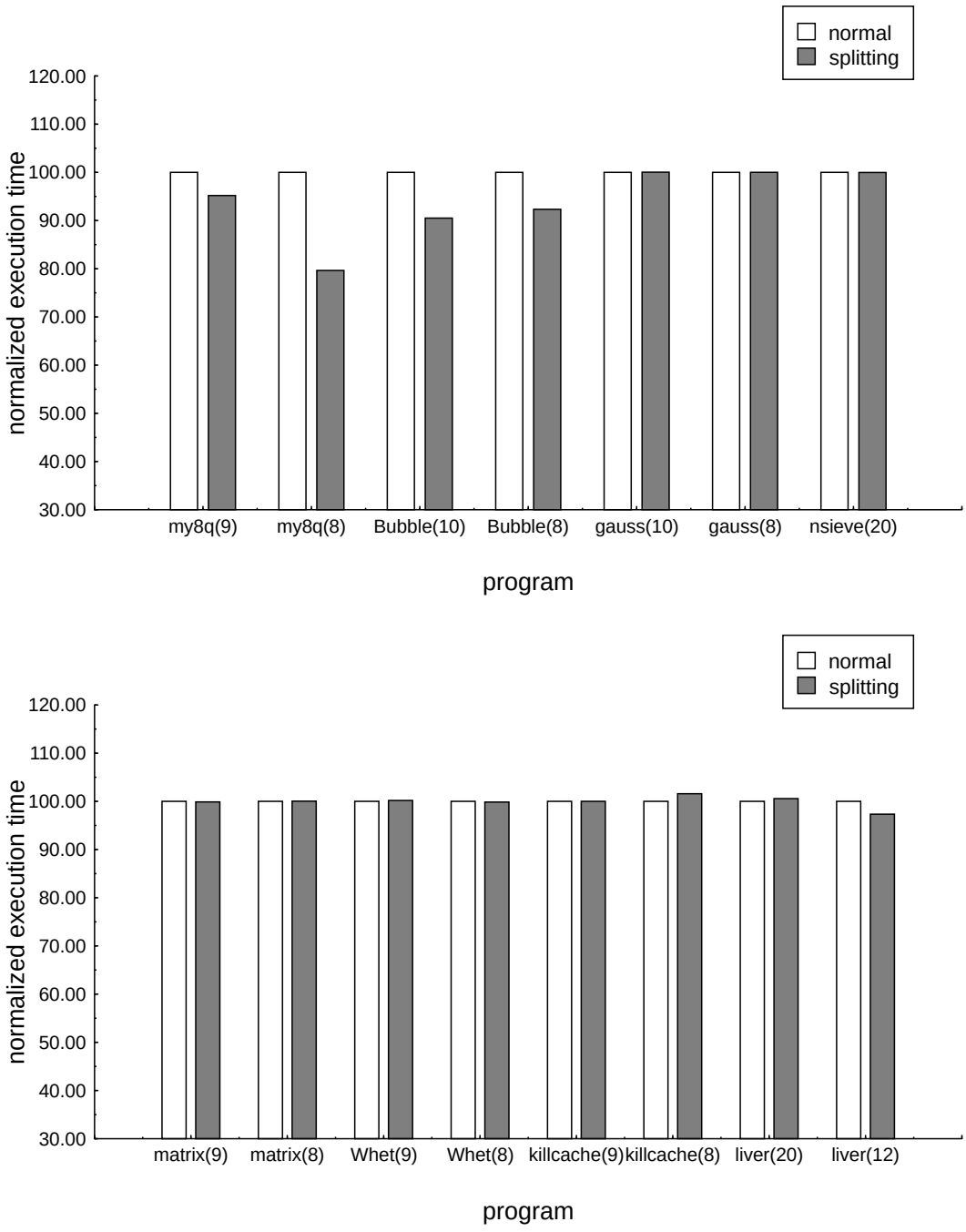

Figure 5.3: Reduction in execution time using normal spill cost heuristic.

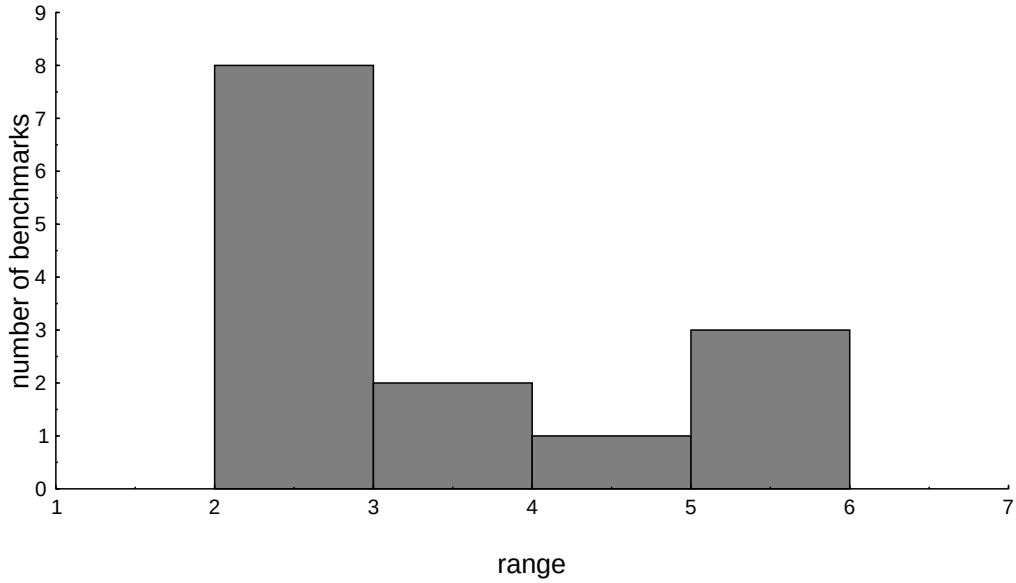

Figure 5.4: Average number of basic blocks per region.

in the number of dynamic memory references for these benchmarks when live range splitting was performed. The average reduction in execution time for all the benchmarks was 2.86%

The average number of basic blocks per low register region was counted through modifying the compiler to output the number of blocks in each region after low register pressure regions were computed. The histogram in Figure 5.4 shows the distribution of the benchmarks with the number of basic blocks per region. The number of basic blocks per region is shown in Table A.4 in the Appendix. There were 8 benchmarks that had an average number of basic blocks per region in a range between 2 and 3. The average number of blocks per region that was between 3 and 4 was met by 3 benchmarks, as was the case for the range between 5 and 6. The was only one benchmark that had an average number of blocks between 4 and 5. Programs with an average number of basic blocks greater than 2 did not correspond to large reductions in spill code. The benchmarks that benefitted the most from live range splitting, *my8q* and *bigBubble*, had an average 2.7 basic blocks per region.

In order to understand why certain benchmarks benefitted from live range splitting, it is

necessary to examine the conditions that are required for live range splitting to occur. It is necessary for at least one of the low register pressure regions in a program to contain more than two basic blocks in order for live range splitting to be effective in reducing the amount of spill code. It is also necessary that a spilled variable be live through most of the function with a number of uses throughout the live range. In other words, a large low register pressure region needs several references to the spilled variable within the region in order to for live range splitting to take place. This can be shown through an analysis of one live range in the *my8q* benchmark which will illustrate why this benchmark had the largest reductions in static spills, dynamic loads, and execution time.

One function, *queens*, in the test case *my8q*, with 9 registers, contains 4 low register pressure regions. One region contains 4 basic blocks. This region, along with the spilled variable *cP* and each block's execution frequency, is shown in Figure 5.5. There are 2 load candidates for this region, B4 and B6, with B6 being chosen based on the profile information. Thus, when live range splitting is performed 1 split-load replaces 4 loads added in the spill-everywhere approach. The conditions described above are met in this function. There is at least one large low pressure region. The spilled variable, *cP*, has a live range that extends from its definition in the entrance block to a final use in a block that is a direct predecessor of the exit block. In addition to this long live range, there are 7 uses of the variable, 6 which are in low register pressure regions. Four of these uses are in the low pressure region described in Figure 5.5.

In contrast, consider the *matrix_mult* benchmark with 9 registers. The *main* function in this program, which will spill a variable, contains 4 low register pressure regions, one which contains 17 basic blocks. However, the variable which spills has a short live range with 2 uses and 2 definitions. One definition occurs at the exit block of the low register pressure region with 17 blocks. Because of this spill location, an opportunity to find a more beneficial store candidate block is not present. There are no blocks in the region that post-dominate the definition. Another low register pressure region constains one use and one definition of the spilled variable. However, this low register pressure region contains only one basic block. It is for these reasons live range splitting had no effect on this test case. The necessary conditions for live range splitting are present, a large low pressure region and a

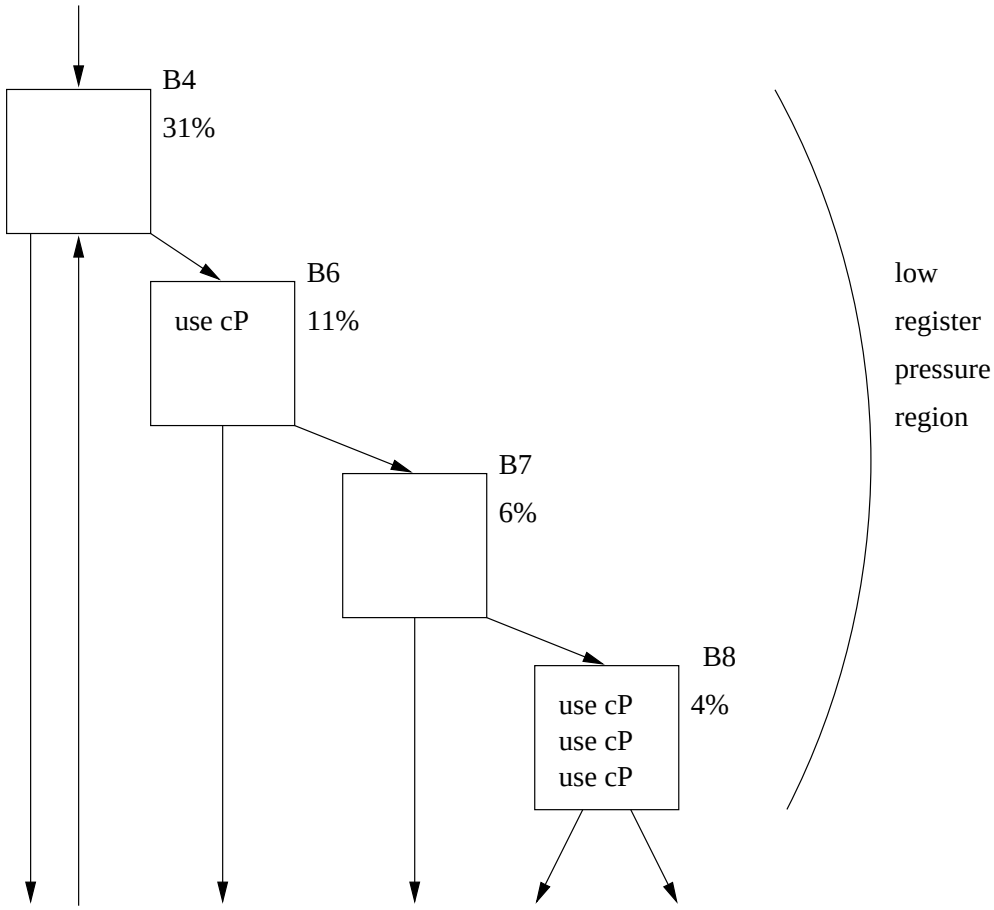

Figure 5.5: Low register pressure region in *my8q* benchmark using normal spill cost heuristic.

region with several references to the spilled variable. However, these conditions are never combined. There does not exist a large low register pressure region with several references to the spilled variable, and live range splitting does not occur.

Live range splitting can take place without substantially increasing the compile time of the program. The is shown in the graphs in Figure 5.6. The register allocation compile time was measured by recording by the user time that elapsed during this phase of the compiler. The graphs represent the normalized results for each benchmark. For example, the benchmark *my8q* with 9 registers actually reduced register allocation compile time by 2.5%.

The average increase in register allocation time was 2.29%. For the benchmarks that had the largest reduction in spill code and execution time there was a decrease in register allocation time. The programs *my8q* and *bigBubble* had an average 3.64% decrease in register allocation time. One possible reason for this is by performing live range splitting, the latter stage of introducing spill code for most references to the spilled variable is eliminated. The extra steps needed for live range splitting had a larger effect on programs with large compile times. For example, the program *livermore*, with 12 registers had the largest increase in compile time, 16.51%.

5.3.2 Using Modified Chaitin's Spill Cost Heuristics

This section graphs the data collected for each benchmark when using a spill-everywhere approach and data for live range splitting that incorporates low register pressure region information into the spill cost heuristics. The modified cost heuristic uses register pressure information when computing a total execution cost for each live range in the interference graph. These graphs show the normalized results for each test. Since a different spill cost heuristic is used with live range splitting than is used with the normal spilling approach, it is possible that different spilled variables are chosen with live range splitting. The results will show that this can have a negative impact on the performance of the program.

The graphs in Figure 5.7 show the normalized results for the reduction in static spills when using live range splitting. The results are similar to the previous results when the normal

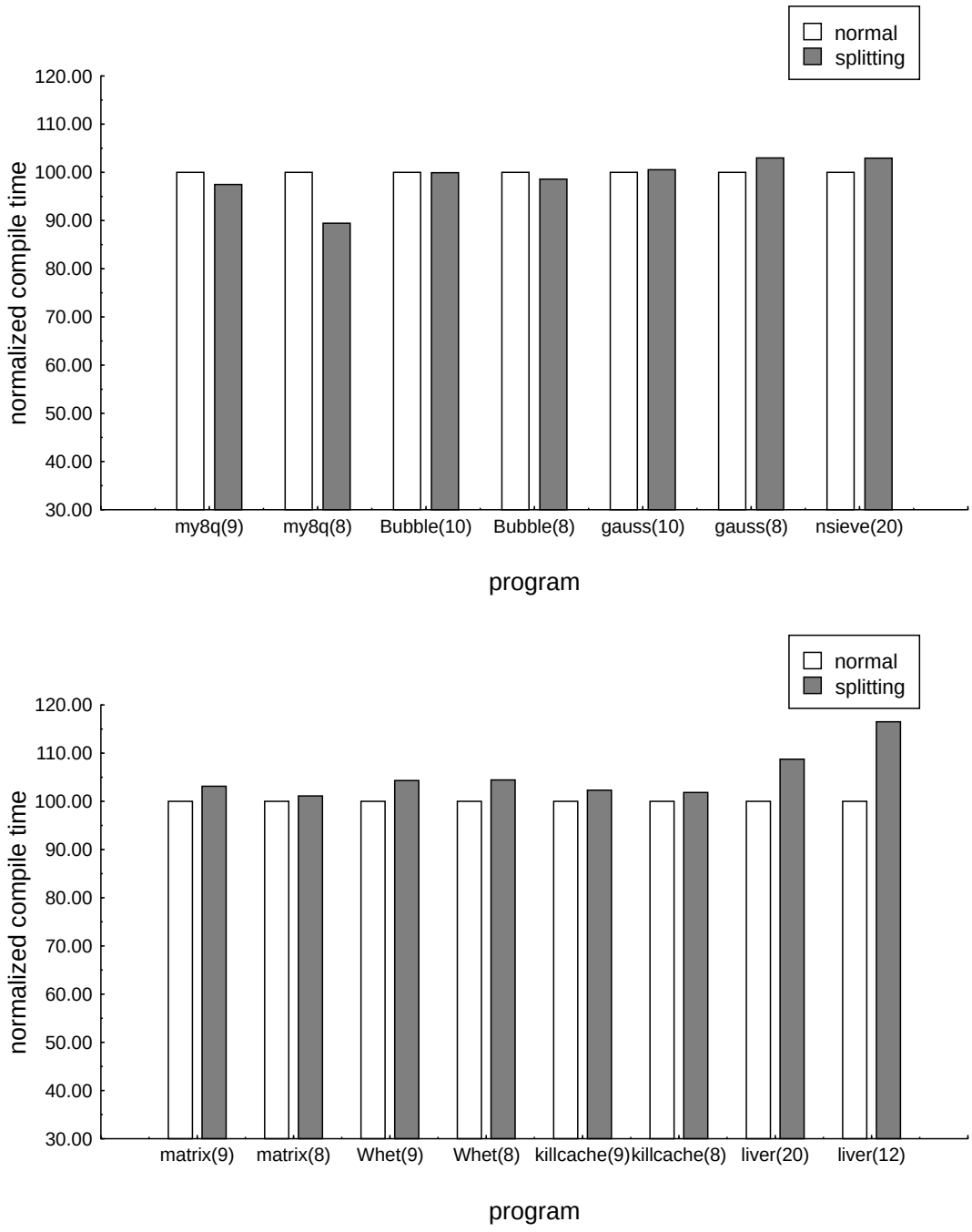

Figure 5.6: Increase in register allocation compile time using normal spill cost heuristic.

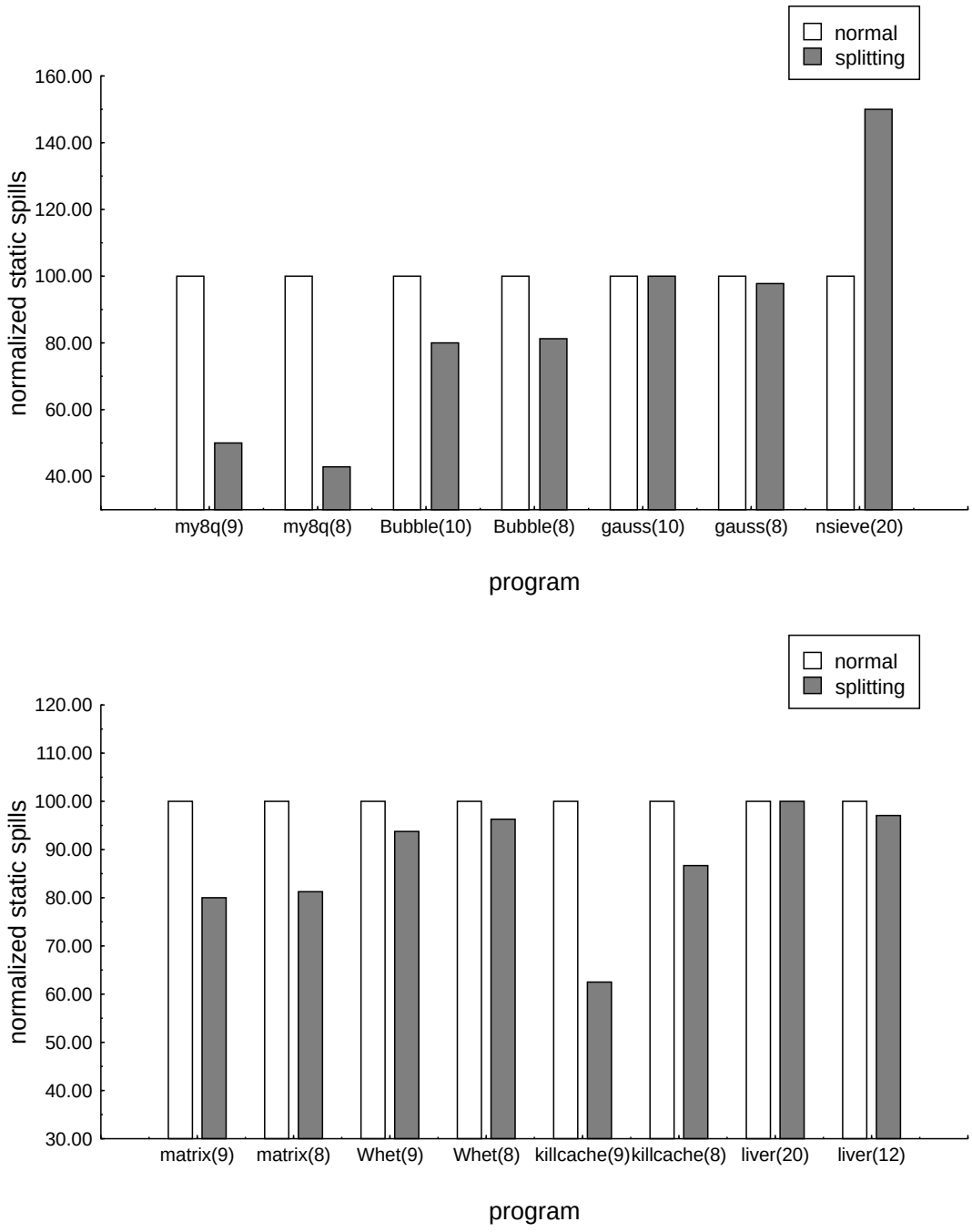

Figure 5.7: Reduction in static spills using modified spill cost heuristic.

spill cost heuristic was used. There are 3 test cases were there was a significant change. The benchmark, *my8q* with 9 registers, reduced static spills by 50% instead of the 60% that occurred with the normal spill cost heuristic. The benchmark *nsieve* increased the number of static spills by 50%. The benchmark *killcache*, with 9 registers reduced static spills by 37.5% when there was no reduction in static spills using normal spill cost heuristics. Each change in the number of static spills was the result of a different variable being chosen to spill than the variable chosen to spill using the normal cost heuristics. A different spill variable was also chosen for the livermore benchmark, with 20 registers, although the total number of static spills did not change.

The different spilled variables chosen in the previous 4 benchmarks resulted in more dynamic loads, and in some cases more dynamic stores, than were executed in the test cases using normal spilling. The graphs in Figure 5.8 show the normalized results for dynamic loads and stores. The benchmark *my8q*, with 9 registers, had the largest increase in dynamic memory references. There was an 8.6% increase in load instructions executed and a 76.99% increase in store instructions executed. The *killcache* benchmark, with 9 registers, had a 24.96% increase in load instructions executed, with a negligible increase in dynamic stores. The other benchmarks had a negligible increase in dynamic loads and stores. For example, the program *nsieve* had 15 more load instructions execute and *livermore*, with 20 registers, had 475,000 more load instructions execute.

This increase in dynamic memory references was reflected in an increase in execution time for the 4 previously mentioned benchmarks. The results for all benchmarks are graphed in Figure 5.9, which shows the normalized results. The *my8q* program, with 9 registers, saw the largest increase with a 14.15% increase in execution time. There was a slight increase in the other 3 benchmarks, *nsieve*, *killcache*, and *livermore*, with an average increase of 4.53% for all 4 benchmarks.

The reason these benchmarks had a worse performance over normal spilling was due to the spilled choice that was made. The modified spill cost heuristic is designed to choose a variable with most of its references in low register pressure regions. This was the case for each new variable chosen to spill in the 4 test cases. However, it is possible that this new spilled variable may execute in a high register pressure region where no spilling was needed

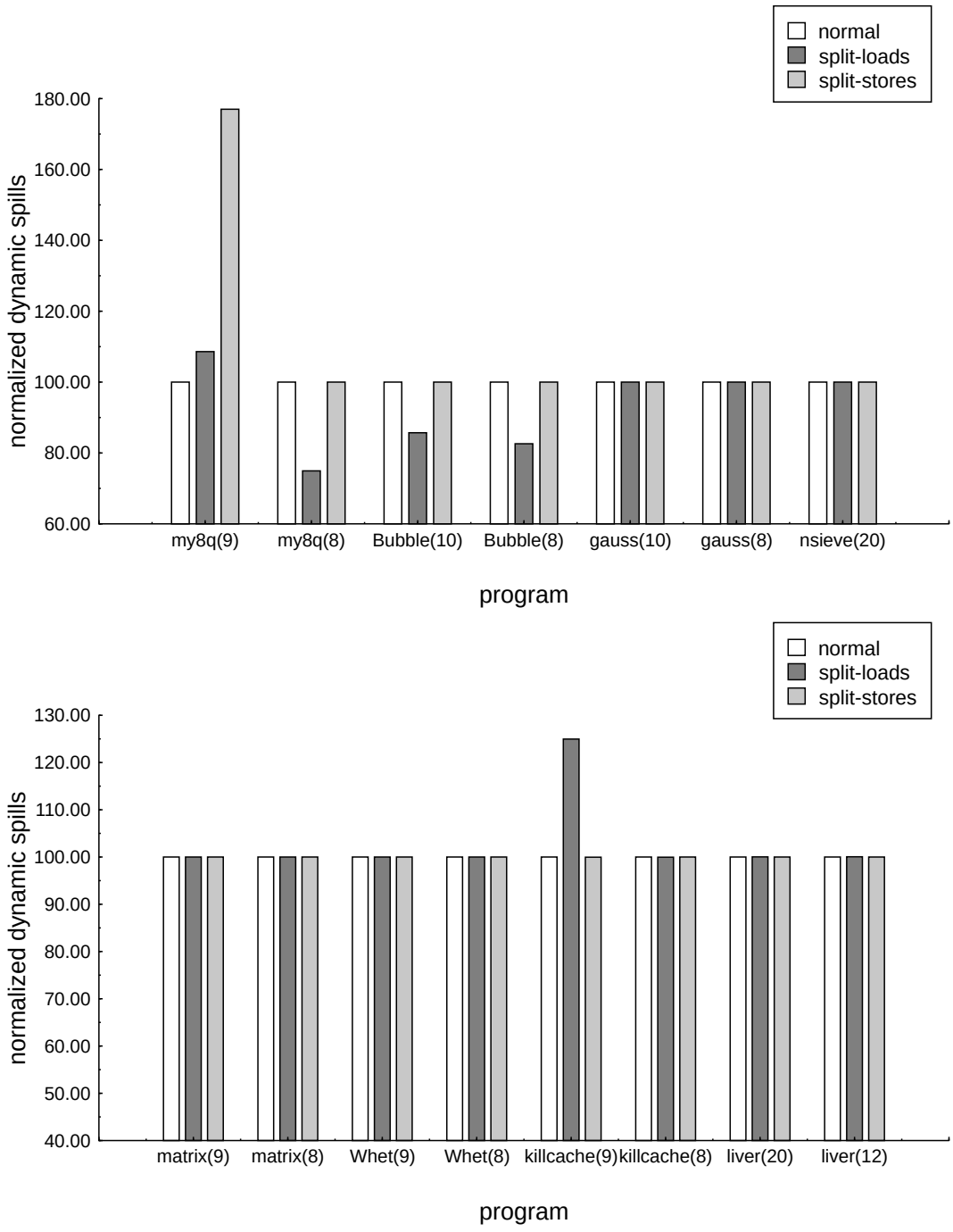

Figure 5.8: Change in dynamic loads and stores using modified spill cost heuristic.

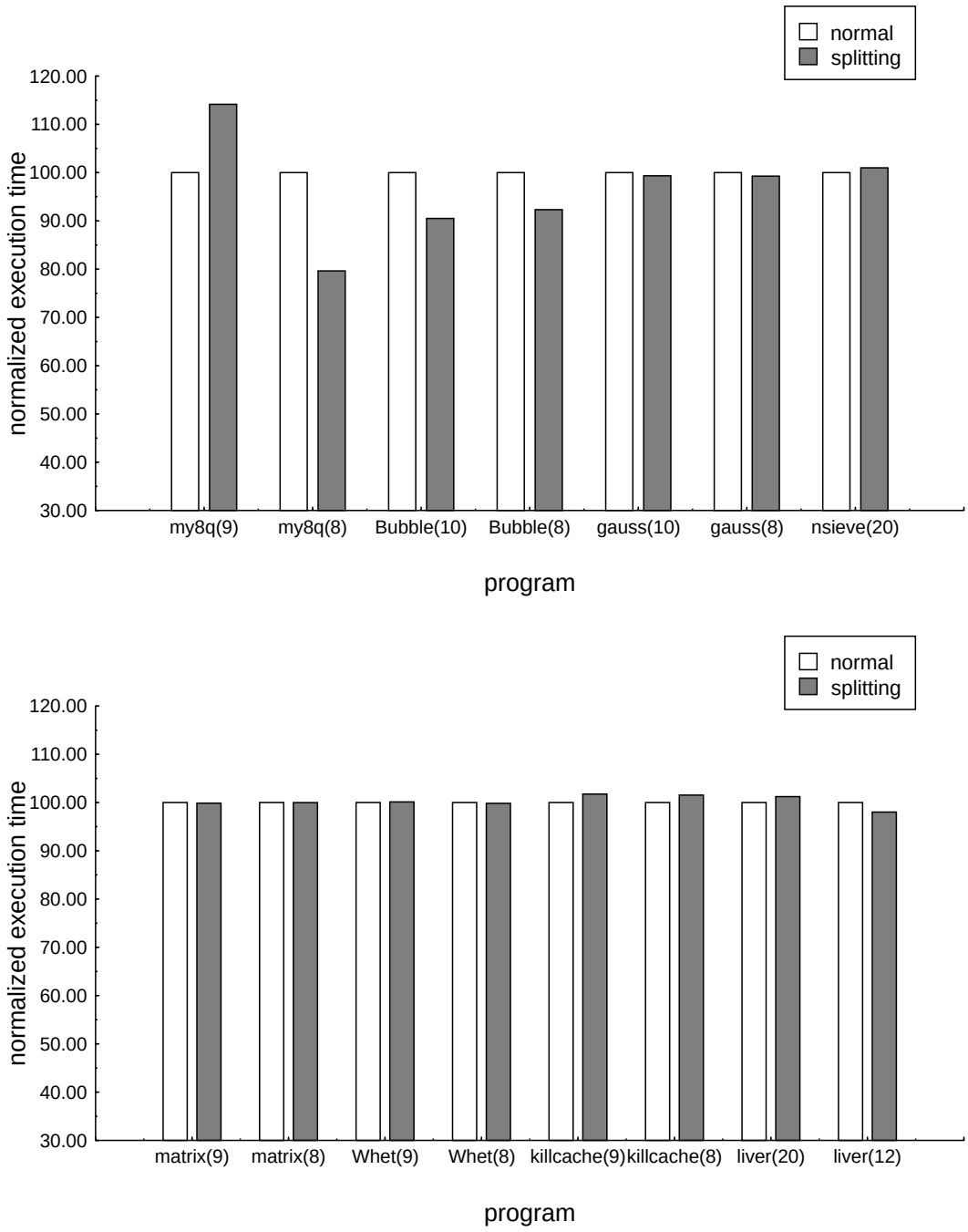

Figure 5.9: Reduction in execution time using modified spill cost heuristic.

using the normal spill cost heuristic. It may also be possible that live range splitting is now performed in a low register pressure region that had no spilled variables using normal spill cost heuristics. In either case, the region where spill code is now being added, using the modified spill cost heuristic, can have a higher execution frequency than previous spill locations.

By examining the *my8q* benchmark with 9 registers, this can be seen. As shown in the previous section, the variable cP was chosen to spill. This variable has references in 3 of the 4 low register pressure regions and splitting is done in one region as was illustrated in Figure 5.5. With the modified spill cost heuristics a new variable, r, is chosen to spill. The heuristic has worked as designed since all of r's references are in low register pressure regions. However, there are uses of r in the one low register pressure region that did not contain any references to variable cP. Live range splitting reduces the number of static spills of r in this region by 50%. A store is also added to this region since r is defined in this region. The worse performance is due to the execution frequency of the load and store candidate block for this low register pressure region. Using normal spill cost heuristics, the load candidate block for variable cP spills has a 3.9% execution frequency. There are no references to r in this block. In the load and store candidate block for variable r the execution frequency is 31%. As a result, more load and store instructions are executed by this benchmark.

Using the modified spill cost heuristics created a large penalty in register allocation compile times. The normalized results are shown in Figure 5.10. The average for all the benchmarks was a 23.28% increase in register allocation time. The reason for this increase is that it is necessary to compute pressure values for each basic block prior to building the interference graph. These values are needed to determine if a block will be a member of a low register pressure region, which is used by the spill cost heuristic. This computation occurs for each register bank, since each bank will build its own interference graph. This increases the register allocation time since register pressure is computed for each basic block for the floating point register bank, for example, even if no floating point variables will spill.

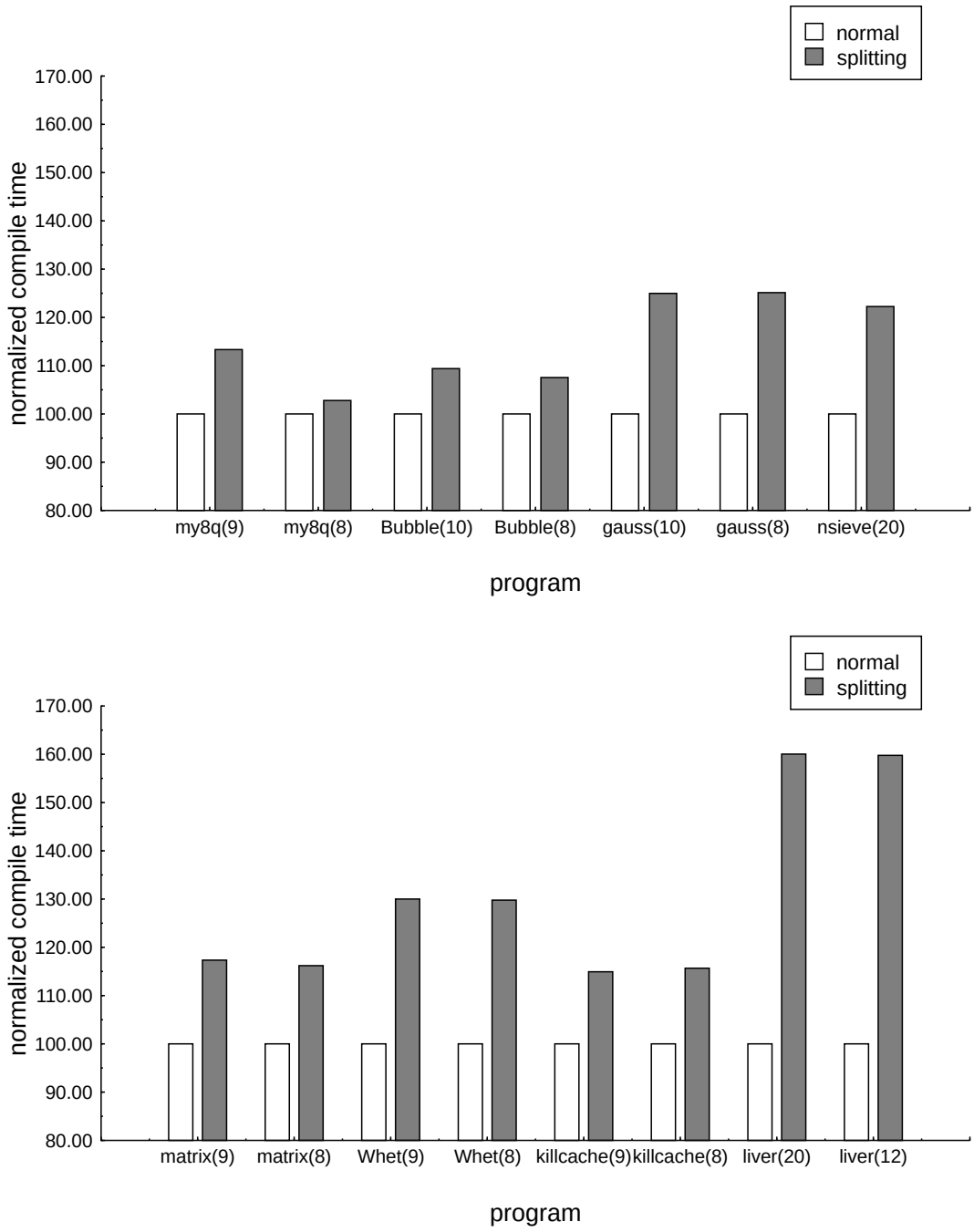

Figure 5.10: Increase in register allocator time using modified spill cost heuristic.

5.4 Lessons Learned

Live range splitting can be an effective method to reduce both the number of static spills generated by the register allocator along with the number of dynamic spills. The Briggs' allocator produces poor code if there are a few high register pressure regions along with a number of constrained live ranges that all have high execution frequencies. When a live range spills there will be a number of memory references added to low register pressure regions. These conditions are taken advantage of by our live range splitting algorithm and the number of memory references in the low register pressure regions is reduced. This reduction also occurs without a substantial penalty in compile times. Live range splitting is not as effective if the register allocator can choose a spill candidate with a low execution frequency. In this case, the opportunities for reducing spill code through live range splitting do not exist.

Using a modified spill cost heuristic with live range splitting was not successful. One reason for this is that the modified spill cost function only considers if a use or definition of a variable occurs in a low register pressure region. However, some spilling is required when the live range is split. The cost heuristic does not take into account the execution frequency of reloading a spill variable into a low register pressure region since this information is not yet available. Since this reload could have a higher execution frequency than the execution frequencies of spill code for a different live range, the modified heuristic will often chose a poor spill candidate. The normal spill cost heuristic is designed to find a live range with a minimum spill cost. If a live range exists that will add only a minimum amount of spill code the potential to find a live range that will add less spill code through live range splitting will not exist. The modified spill cost heuristic was also unsuccessful in terms of compiles times, with substantial increases seen in the benchmarks.

Chapter 6

Conclusions and Future Work

Current, state-of-the-art compilers use Briggs' graph coloring algorithm for register allocation since this provides an efficient method for mapping program variables to machine registers. However, when a live range cannot be allocated a register, spill code is added to the program for each use and definition of the variable in the live range. Several algorithms were examined that attempt to reduce the amount of spill code through live range splitting. Notably, the work done by Bergner revealed that live range splitting can reduce a significant amount of the spill code added by the register allocator and achieve a reduction in the execution time of the program. The disadvantage to Bergner's interference region spilling algorithm was a substantial increase in compile time.

Our algorithm for live range splitting can also reduce the amount of spill code without the large increase in compile time. The algorithm identifies low register pressure regions and limits the amount of spill code in those regions. Identifying the low register pressure regions is efficient since it is based on the near linear algorithm for computing symbolic covers developed by Reif and Tarjan. Live range splitting was implemented in the Rocket compiler in order to test our algorithm's ability in reducing spill code and improving a program's performance.

The benchmarks that were tested with this algorithm demonstrated that significant reductions in the amount of spill code and execution time can be obtained. Although not

every benchmark saw a reduction in spill code, the algorithm will perform as well as the spill-everywhere approach of a Briggs' register allocator in the worst case. Live range splitting was also performed without any significant increase in compile times. Using the low register pressure region information, used by our live range splitting algorithm, in the spill cost heuristics was also examined. The results for the benchmarks using the new spill cost heuristic were not successful since the location of spill code in a load candidate block or store candidate block in a low register pressure region is not available to the heuristic.

6.1 Future Work

Additional work needs to done in order to determine the range of effectiveness of our live range splitting algorithm. In particular, it would be necessary to examine programs with high register pressure that will spill without artificially lowering the number of available machine registers. This will determine if the conditions described as necessary for live range splitting exist in these programs. It also needs to be determined if the average number of basic blocks per low register pressure region is larger than was observed in the benchmarks tested in this thesis. If average number of blocks is similar to the numbers observed here, a method for combining small low register pressure regions could be explored. It was often the case in the benchmark programs that many of the low register pressure regions contained basic blocks that bordered other low register pressure regions. It would be worthwhile to see there are conditions where two regions can be combined without the need for additional load or store instructions in the new region.

Appendix A

Experimental Data

This section contains the data collected for each benchmark using both the normal spill cost heuristic and the modified spill cost heuristic. The number of registers listed in the tables represents the number of available integer and floating point registers used for that benchmark. Each table also includes the percent reduction for the number of spills or execution time. Percent increase is shown for register allocation time.

		Normal Spilling			Live Range Splitting					Percentage Reduction
Program	#Regs	Loads	Stores	Total	Loads	Stores	Split-Loads	Split-Stores	Total	
my8q	9	9	1	10	1	1	2	0	4	60
my8q	8	23	5	28	1	3	6	2	12	57.14
bigBubble	10	4	1	5	2	1	1	0	4	20
bigBubble	8	11	5	16	5	4	3	1	13	18.75
gauss	10	1	2	3	1	2	0	0	3	0
gauss	8	29	16	45	28	15	0	1	44	2.22
nsieve	20	1	1	2	1	1	0	0	2	0
matrix_mult	9	3	2	5	1	1	1	1	4	20
matrix_mult	8	10	6	16	4	3	3	3	13	18.75
myWhetstone	9	13	3	16	11	2	1	1	15	6.25
myWhetstone	8	20	7	27	18	6	1	1	26	3.7
killcache	9	6	2	8	6	2	0	0	8	0
killcache	8	12	3	15	10	2	0	1	13	13.33
livermore	20	13	13	26	10	13	3	0	26	0
livermore	12	18	16	34	13	15	4	1	33	2.94

Table A.1: Static spills using normal spill cost heuristics - 14.87% average reduction.

		Normal Spilling		Live Range Splitting		Percentage Reduction	
Program	#Regs	Loads	Stores	Loads	Stores	Load	Store
my8q	9	1902420001	408340001	1586660001	408340001	16.6	0
my8q	8	2958340001	722760002	2217300001	722760002	25.05	0
bigBubble	10	5597346938	1198713471	4797406937	1198713471	14.29	0
bigBubble	8	9196984755	1998773468	7597024755	1998773468	17.4	0
gauss	10	275221280	45730059	275221280	45730059	0	0
gauss	8	277004057	46127113	277003545	46127113	$1.8e-04$	0
nsieve	20	459294647	355259711	459294647	355259711	0	0
matrix_mult	9	807929351	135004674	807929351	135004674	0	0
matrix_mult	8	942934023	135267843	942934023	135267843	0	0
myWhetstone	9	2336482534	522454542	2336482534	522454542	0	0
myWhetstone	8	2344334104	530306112	2344334104	530306112	0	0
killcache	9	539761154	1311234	539761154	1311234	0	0
killcache	8	674767877	1311235	674504708	1311235	0.04	0
livermore	20	1614195203	614750001	1614195203	614750001	0	0
livermore	12	1614497703	614755002	1614870203	614755002	-0.02	0

Table A.2: Dynamic spills using normal spill cost heuristics - 4.89% average reduction of loads.

		Normal Spilling	Live Range Splitting	Percentage Reduction
Program	#Regs	Seconds	Seconds	%
my8q	9	40.4936	38.5474	4.81
my8q	8	51.4322	40.9645	20.35
bigBubble	10	87.4749	79.1706	9.49
bigBubble	8	130.078	120.106	7.67
gauss	10	12.1081	12.1125	-0.04
gauss	8	11.9427	11.9434	-0.006
nsieve	20	82.4705	82.4499	0.02
matrix_mult	9	29.9664	29.9297	0.12
matrix_mult	8	31.716	31.7277	-0.04
myWhetstone	9	48.7622	48.8483	-0.18
myWhetstone	8	48.9823	48.9075	0.15
killcache	9	32.9122	32.9135	-0.004
killcache	8	32.9748	33.496	-1.59
livermore	20	33.6505	33.8408	-0.56
livermore	12	34.0668	33.1596	2.66

Table A.3: Execution time using normal spill cost heuristics - 2.86% average decrease.

		Average Blocks Per Region
Program	#Regs	
my8q	9	2.5
my8q	8	3.14286
bigBubble	10	2.5
bigBubble	8	2.5
gauss	10	2.33333
gauss	8	2.06667
nsieve	20	5.2
matrix_mult	9	5.25
matrix_mult	8	2.5
myWhetstone	9	2.54167
myWhetstone	8	2.30769
killcache	9	3.28571
killcache	8	3.28571
livermore	20	4.5
livermore	12	5.06667

Table A.4: Average number of basic blocks per low register pressure region - 3.265 blocks.

		Normal Spilling	Live Range Splitting	Percentage Increase
Program	#Regs	Seconds	Seconds	
my8q	9	3.17688	3.09685	-2.52
my8q	8	3.99379	3.57216	-10.56
bigBubble	10	1.38202	1.38104	-0.07
bigBubble	8	1.52646	1.50499	-1.41
gauss	10	19.5669	19.6742	0.55
gauss	8	27.7945	28.6251	2.99
nsieve	20	45.7003	47.0471	2.95
matrix_mult	9	3.39843	3.50482	3.13
matrix_mult	8	3.65707	3.69806	1.12
myWhetstone	9	189.333	197.504	4.32
myWhetstone	8	192.69	201.246	4.44
killcache	9	3.42381	3.50286	2.31
killcache	8	3.55557	3.62096	1.84
livermore	20	860.85	936.127	8.74
livermore	12	873.467	1017.66	16.51

Table A.5: Register allocation time using normal spill cost heuristics - 2.29% average increase.

		Normal Spilling			Live Range Splitting					Percentage Reduction
Program	#Regs	Loads	Stores	Total	Loads	Stores	Split-Loads	Split-Stores	Total	
my8q	9	9	1	10	0	1	3	1	5	50
my8q	8	23	5	28	1	3	6	2	12	57.14
bigBubble	10	4	1	5	2	1	1	0	4	20
bigBubble	8	11	5	16	5	4	3	1	13	18.75
gauss	10	1	2	3	1	2	0	0	3	0
gauss	8	29	16	45	28	15	0	1	44	2.22
nsieve	20	1	1	2	1	1	1	0	3	-50
matrix_mult	9	3	2	5	1	1	1	1	4	20
matrix_mult	8	10	6	16	4	3	3	3	13	18.75
myWhetstone	9	13	3	16	11	2	1	1	15	6.25
myWhetstone	8	20	7	27	18	6	1	1	26	3.7
killcache	9	6	2	8	4	0	0	1	5	37.5
killcache	8	12	3	15	10	2	0	1	13	13.33
livermore	20	13	13	26	5	13	8	0	26	0
livermore	12	18	16	34	8	15	9	1	33	2.94

Table A.6: Static spills using modified spill cost heuristics - 13.37% average reduction.

		Normal Spilling		Live Range Splitting		Percentage Reduction	
Program	#Regs	Loads	Stores	Loads	Stores	Load	Store
my8q	9	1902420001	408340001	2066020001	722740001	-8.6	-76.99
my8q	8	2958340001	722760002	2217300001	722760002	25.05	0
bigBubble	10	5597346938	1198713471	4797406937	1198713471	14.29	0
bigBubble	8	9196984755	1998773468	7597024755	1998773468	17.4	0
gauss	10	275221280	45730059	275221280	45730059	0	0
gauss	8	277004057	46127113	277003545	46127113	$1.8e-04$	0
nsieve	20	459294647	355259711	459294662	355259711	$-3.3e-06$	0
matrix_mult	9	807929351	135004674	807929351	135004674	0	0
matrix_mult	8	942934023	135267843	942934023	135267843	0	0
myWhetstone	9	2336482534	522454542	2336482534	522454542	0	0
myWhetstone	8	2344334104	530306112	2344334104	530306112	0	0
killcache	9	539761154	1311234	674502147	1310722	-24.96	.04
killcache	8	674767877	1311235	674504708	1311235	.04	0
livermore	20	1614195203	614750001	1614670203	614750001	-0.03	0
livermore	12	1614497703	614755002	1615345203	614755002	-0.05	0

Table A.7: Dynamic spills using modified spill cost heuristics - 1.54% average reduction of loads, -5.13% average increase in stores.

		Normal Spilling	Live Range Splitting	Percentage Reduction
Program	#Regs	Seconds	Seconds	%
my8q	9	40.4936	46.2215	-14.15
my8q	8	51.4322	40.9561	20.37
bigBubble	10	87.4749	79.1618	9.5
bigBubble	8	130.078	120.095	7.67
gauss	10	12.1081	12.0275	0.67
gauss	8	11.9427	11.8551	0.73
nsieve	20	82.4705	83.2865	-0.99
matrix_mult	9	29.9664	29.926	0.13
matrix_mult	8	31.716	31.7115	0.01
myWhetstone	9	48.7622	48.8232	-0.13
myWhetstone	8	48.9823	48.8996	0.17
killcache	9	32.9122	33.4883	-1.75
killcache	8	32.9748	33.4823	-1.54
livermore	20	33.6505	34.063	-1.23
livermore	12	34.0668	33.3943	1.97

Table A.8: Execution time using modified spill cost heuristics - 1.43% average decrease.

		Normal Spilling	Live Range Splitting	Percentage Increase
Program	#Regs	Seconds	Seconds	
my8q	9	3.17688	3.60046	13.33
my8q	8	3.99379	4.10506	2.79
bigBubble	10	1.38202	1.51183	9.39
bigBubble	8	1.52646	1.64163	7.54
gauss	10	19.5669	24.4537	24.97
gauss	8	27.7945	34.7798	25.13
nsieve	20	45.7003	55.8691	22.25
matrix_mult	9	3.39843	3.98794	17.35
matrix_mult	8	3.65707	4.2495	16.2
myWhetstone	9	189.333	246.146	30
myWhetstone	8	192.69	250.063	29.77
killcache	9	3.42381	3.93523	14.94
killcache	8	3.55557	4.11286	15.67
livermore	20	860.85	1377.65	60.03
livermore	12	873.467	1395.55	59.77

Table A.9: Register allocation time using modified spill cost heuristics - 23.28% average increase.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers, Principles, Techniques and Tools. Addison-Wesley Publishing Company, pages 632 - 633, 1986.

[2] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill Code Minimization via Interference Region Spilling. *Proceedings of the ACM 1997 Conference on Program Language Design and Implementation*, pages 287-295, May 1997.

[3] Preston Briggs. Register Allocation via Graph Coloring PhD thesis, Rice University, April 1992.

[4] Preston Briggs, Keith D.Cooper, Ken Kennedy, and Linda Torczon. Coloring Heuristics for Register Allocation. *Proceedings of the ACM SIGPLAN 1989 Conference on Program Language Design and Implementation*, pages 275 - 284, June 1989.

[5] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, pages 428-455, Vol. 16, No. 3, May 1994.

[6] David Callahan and Brian Koblenz. Register Allocation via Hierarchical Graph Coloring. *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 192 - 203, June 1991.

[7] Gregory J Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register Allocation via Coloring. *Computer Languages* Vol 6, pages 47 - 57, January 1981.

[8] G.J. Chaitin. Register Allocation and Spilling via Graph Coloring. *Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction*, pages 98 -105, November 1982.

[9] Fred C. Chow and John L. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4, pages 501 - 536, October 1990.

[10] Thomas H. Cormen, Charles E. Leiserson and Ronald L.Rivest. Introduction to Algorithms. The MIT Press, 1990.

[11] H.J. Curnow and B.A. Wichman. A Synthetic Benchmark *Computer Journal*, Vol. 19, No. 1, February 1976.

[12] Digital Equipment Corporation. ATOM Reference Manual. Digital Equipment Corporation, December 1993.

[13] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Copper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fisher, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, Gilbert M. Wolrich. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, Vol. 7, No. 1, pages 119 - 135, 1995.

[14] Michael R. Garey and David S. Johnson. Computers and Intractability, A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, page 191, 1979.

[15] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, Vol 1, NO. 1, pages 121-141, July 1979.

[16] Steven S. Muchnik. Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers, Inc., 1997.

[17] John H. Reif and Robert E. Tarjan. Symbolic Program Analysis in Almost-Linear Time *SIAM Journal of Computing*, Vol. 11, No. 1, pages 81 - 93, February 1981.

[18] P.H. Sweany. Inter-Block Code Motion without Copies. PhD thesis, Colorado State University, 1992.

[19] Philip H. Sweany and Steven J. Beaty. Overview of the ROCKET Retargetable Compiler. Technical Report CS-94-01, Department of Computer Science, Michigan Technological University, Houghton, MI, January 1994.

[20] Qunyan Wu. Register Allocation via Hierarchical Graph Coloring. Masters thesis, Michigan Technological University, 1996.