THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS & SCIENCES

REDUCING THE COST OF COMPARISONS WITHIN CONDITIONAL TRANSFERS OF CONTROL

By

WILLIAM C. KREAHLING

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 2005

The members of the Committee approve the dissertation of William C. Kreahling defended on July 12, 2005.

David Whalley
Professor Directing Dissertation

Steve Bellenot
Outside Committee Member

Robert van Engelen
Committee Member

Ashok Srinivasan
Committee Member

Xin Yuan
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

This work is dedicated to Dr. Alice McRae, the one who started me down this road in the first place. And to my family, who supported me through the good times and the bad. It's those bad ones that count.

ACKNOWLEDGMENTS

I would like to thank Steve Hines, Prasad Kulkarni and Clint Whaley, for all your support, technical and moral. I think you guys understand the heart attack! I would also like to thank my advisor, Dr. David Whalley. This work could not have been done without his help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A significant percentage of execution cycles are devoted to performing conditional transfers of control, which occur frequently, cause pipeline flushes when mis-predicted and can prevent other code improving transformations from being applied. The work required for each conditional transfer of control can be broken into three portions: the comparison, the calculation of the branch target address and the actual transfer of control. Most of the research investigating the reduction of costs associated with these instructions have focused on the calculation of the branch target address or the actual transfer of control, overlooking the comparison portion.

In this dissertation we propose several techniques to reduce the costs associated with the comparison portion of conditional transfers of control. The first technique attempts to merge multiple conditions, avoiding the execution of comparison and branch instructions. The other two methods decouple the definition of the values to be compared with the actual comparison itself. In the second technique the comparison becomes an effect that implicitly occurs whenever specified registers are assigned values, avoiding the execution of a separate comparison instruction. The third technique uses a similar concept, but instead of implicit comparisons, it uses a single comparison and branch instruction in conjunction with a comparison specification. This technique attempts to capture the advantages of common branching techniques used today, without the traditional disadvantages of using a single instruction to perform both the comparison and branch.

CHAPTER 1

INTRODUCTION

The goal of this research is to reduce the costs associated with conditional transfers of control. Conditional transfers of control can be typically broken into three separate portions. First, a comparison or boolean test between two values is performed. The result of the comparison is typically stored in another register (condition code, general purpose, or predicate) and will later be accessed by a branch instruction. Second is the calculation of the branch target address. The branch target address is typically encoded within the branch instruction, often as a displacement. If the result of the comparison indicates the branch is *taken*, then the address of the next instruction to be fetched must be calculated. Third, the branch instruction itself indicates where the actual transfer of control takes place. The branch instruction accesses the result of the comparison instruction and the next instruction is fetched. The next instruction is either the next sequential instruction or the instruction located at the branch target address. There has been much research conducted with the goal of reducing the costs associated with performing conditional transfers of control. However, the majority of this research has focused on reducing the cost of the calculation of the branch target address and/or the actual transfer of control, while the comparison portion has been largely overlooked.

1.1 Separate Compare and Branch Instructions

Arguably the most common method used to perform conditional transfers of control uses separate comparison and branch instructions. One advantage with this method is freedom to encode the comparison and branch instructions, since two separate instructions are used. When the branch instruction is reached, the execution of the program does not stall waiting for the outcome of the comparison to be known before fetching the next instruction to execute on most processors. Instead, a prediction is made and an instruction is fetched. If the prediction was found to be incorrect when the outcome of the comparison is known, then the pipeline is flushed and instructions from the correct path are fetched. Most branch prediction methods used today are accurate enough that this mis-prediction penalty is infrequently assessed. A disadvantage with this method is there are two separate instructions that need to be fetched and decoded to perform the conditional transfer of control. Although execution does not stall when the branch is reached because of branch prediction, execution on an in-order pipeline will stall when the comparison instruction is reached and the values involved in the comparison are not ready, meaning that the branch instruction following the comparison is stalled as well.

1.2 Single Compare and Branch Instruction

Another way of performing conditional transfers of control is to use a single instruction that performs both the comparison and the branch. Using this method the information needed for the comparison is encoded within the branch instruction itself. Thus, only a single instruction is fetched and decoded for each conditional transfer of control. However, to encode all this information within the bits allocated for a single instruction, fewer bits are typically designated to specify the branch target address. This limits the range of branch target displacements. Likewise comparisons to an arbitrary constant sometimes cannot be made.

1.3 New Techniques Focusing on the Comparison

This dissertation presents three different techniques that reduce the costs associated with conditional transfers of control. Unlike the majority of research in this area, these three techniques focus on the comparison portion of a conditional transfer of control. The first technique tries to reduce the number of comparisons needed in a program by testing multiple conditions at once. A condition is a test whose outcome evaluates to *true* or *false*. The second technique decouples the definition of the values to be compared with the actual comparison itself. The comparison becomes an implicit effect that occurs when values are assigned to specified registers. The third technique also decouples the definition of the values involved in the comparison from the actual comparison, using a single comparison and branch instruction in conjunction with an instruction that defines the comparison that will take place. This technique minimizes the disadvantages associated with other techniques that use a single instruction to perform both a comparison and a branch.

1.4 Organization of Dissertation

The remainder of this dissertation is organized in the following manner: Chapter 2 presents an overview of previous work to reduce costs associated with conditional transfers of control. Chapter 3 presents our technique that reduces the cost of traditional transfers of control by testing multiple conditions at once. Chapter 4 discusses the other two techniques, both of which involve a new instruction called a *comparison specification*. In this section we contrast the two techniques and evaluate experimental results from both. Chapter 5 discusses ways that all three techniques might be improved. Finally, Chapter 6 summarizes our findings and presents our research contributions.

CHAPTER 2

RELATED WORK

Research into eliminating conditional transfers of control or reducing the costs associated with conditional transfers of control is an ongoing area of research. This section looks at several techniques that have been developed in the past that have been used to combat the problem of delays associated with branch instructions.

2.1 Delayed Branches

A very simple and straightforward technique to reduce the cost of stalls from *taken* or *not-taken* branches is *delayed branches.* In an architecture that supports delayed branches, one or more instructions immediately following the branch instruction are always executed regardless of whether or not the branch is taken. The slots immediately following the branch instruction are called *delay slots.*

A critical component to the success of delayed branches is the ability of the compiler to fill the delay slots with instructions that can safely be executed regardless of the outcome of the branch. If no instruction can be found that will safely execute in a delay slot, then the compiler will fill the slot with a *no-op* instruction. Moving an instruction from the code *before* the branch is always a useful way to fill a delay slot, as code that occurs before the branch needs to be executed no matter the outcome of the branch. However, moving instructions from either the *taken* or *not-taken* paths following the branch may or may not perform useful work. To move an instruction from either the *taken* or *not-taken* paths the instruction must be safe to move to a delay slot. For example, if an instruction is moved from the *taken* flow of control, it must be safe and legal for that instruction to execute if the branch resolves to *not-taken*, and vice-versa. Measurements have shown that a *C* optimizing compiler is able to fill between 40% and 60% of delay slots following conditional transfers of control and 90% following unconditional transfers of control [1]. The more delay slots following a branch instruction, the harder it is for the compiler to fill all the slots with instructions that perform useful work.

In some architectures the instructions in the delay slots are able to be conditionally nullified depending on the outcome of the branch. This enables the compiler to choose more instructions to fill the delay slots. An instruction moved from the *taken* path and used to fill a delay slot, no longer has to be safe or legal to execute if the branch is *not-taken.* An instruction in the delay slot will execute if the branch condition resolves a certain way and will be nullified if the branch resolves in the opposite manner. As multiple issue machines rise in popularity, delayed branches are falling out of favor, in part due to the difficulty of filling the delay slots with useful instructions.

2.2 Branch Prediction

The instruction that sequentially follows a branch may not be the next instruction to be executed. If a branch is taken, then the next instruction to be executed is at the address denoted by the branch target address. The flow of control of a program forks at a conditional branch instruction and a decision must be made about which instruction to execute next. In a pipelined machine, if we do not fetch an instruction until we know what instruction executes next, then the CPU will sit idle wasting time and resources. The process of deciding which instruction to execute before we know the outcome of the branch is called *branch prediction.*

2.2.1 Branch Prediction Buffer

Hardware prediction successfully reduces branch costs when it accurately predicts the direction a branch will take [2]. In a *branch prediction buffer* (BPB) the low order bits of an instruction are used to index into a table. Prediction bits are used from this table to predict the outcome of a branch. Each time a branch is executed the table is updated. Several different schemes can be used for the BPB. The simplest scheme uses one bit to predict the outcome of a branch. If the branch is predicted *taken*, and resolves to *not-taken* then the bit is flipped, and vice-versa. However, this simple scheme causes branches to be mispredicted unnecessarily when a loop is executed several times in close proximity. Modifying the BPB to use a two bit prediction scheme solves the problem.

2.2.2 Correlating Predictors

While the two bit scheme used in the BPB is an improvement over the one bit scheme, only the recent behavior of *one* branch is taken into consideration when trying to predict the behavior of a branch. If the behavior of other branches are taken into consideration when predicting branch outcomes, then more accurate predictions can be made. *Correlating predictors* (two-level predictors) use the behavior of multiple instances of previous branches to help make predictions.

Correlating predictors can be generalized to use the behavior of the last m branches to choose from among 2^m predictors, each having n bits. This generalized form of correlating predictors is called an (m, n) predictor. A shift register can be used to record the behavior of the last m branches, where each bit in the shift register represents whether one of the previously executed branches was *taken* or *not-taken*.

2.2.3 Tournament Predictors

Tournament or hybrid predictors simply combine two or more prediction methods together, taking into account that different prediction methods may work better for different types of branches. In work done by Scott McFarling [3], two types of predictors are combined and an array of two bit saturating counters is used to determine which branch prediction method to use. McFarling conducted experiments, in which the two prediction methods were *bimodal* and *gshare* prediction. In these experiments the tournament predictor consistently outperformed each technique performed in isolation [3].

2.2.4 Markov Predictors

Recent work in the field of branch prediction [4, 5] applies techniques common in the field of data compression to the problem of branch prediction. It is shown in work done by Chen, et. al., that *correlating* or *two-level* predictors are a simplification of an optimal predictor used in data compression called *predication by partial matching* (PPM). Experiments indicate that PPM outperforms two-level predictors, however it may not be feasible to build optimal predictors given the current level of technology [4].

2.2.5 Neural Methods for Branch Prediction

Neural methods of branch prediction try to find a simple neural method to use as an alternative to the commonly used two bit counters. One such neural method uses a *perceptron* predictor to replace the two bit counter [6]. A longer history length can be taken into account as part of the prediction, when using perceptron predictors, as the hardware resource for this method scales linearly, rather than exponentially, with the history length. Thus, perceptron predictors can consider much larger histories with the same resources as the two bit predictor.

A perceptron is a learning device that takes a set of input values and combines them with a set of weights (learned though training) to produce an output value. In branch prediction, the weights represent the correlation between the current branch and the past behavior of other branches. The weight is incremented when a branch is *taken* and the weight is decremented when the branch is *not-taken*.

Experiments using perceptron predictors, carried out through simulation, showed improvements over McFarling style hybrid predictors [6], although the complexity of both the hardware and the computation is increased when using perceptron predictors. Thus, questions remain about the feasibility of realistically implementing this method without increasing the cycle time of the machine.

2.3 Branch Target Buffer

Even when hardware prediction successfully reduces the branch cost with an accurate branch prediction a delay may still occur while the branch target address is calculated. The BTB acts as a small cache of branch target addresses, which when given the address of a branch will return the actual target address.

Each entry of a BTB typically consists of a tag containing a portion of a branch instruction's address plus the branch's target address. The next several instructions following the branch target may also be located within the buffer. If the branch instruction's address is not in the BTB, then a prediction of *not-taken* occurs. If the prediction is incorrect, then the pipeline is stalled while the correct address is fetched. If a branch is predicted to be *taken* and resolves to *not-taken*, then the tag corresponding to that branch is deleted from the BTB. On the other hand, if a branch is predicted *not-taken* and resolves to *taken*, then the branch target address is inserted in the BTB. Branch target buffers and branch prediction buffers are often combined to gain the benefits of more accurate prediction methods.

2.4 Branch Registers

Another technique to reduce the cost of executing a branch uses traditional registers to hold the branch target address and the instructions corresponding to it [7]. In this research two machines were designed and emulated. The baseline version had 32 general-purpose registers and the other had 16 general-purpose registers and 16 registers that were used for branching. The calculation of the branch target address is separated from the instruction that performs the transfer of control. By exposing the computation of the branch target address to the compiler, this technique provides new opportunities for optimization. In many cases the calculation of the branch target address may be moved out of loops, thereby incurring the cost of the calculation once per loop.

2.5 Predication

Predicated execution is the conditional execution of an instruction based upon the value of a boolean source operand, called the *predicate*. Predicated instructions are fetched regardless of the value of their predicate. Instructions whose predicate value is *true* execute normally. Instructions whose predicate value is *false* are nullified.

There are multiple benefits when an architecture has support for predicated execution. First, reducing the number of a branches reduces the need to sustain multiple branches per cycle. Second, eliminating frequently mispredicted branches leads directly to the substantial elimination of branch mispredictions. Finally, predicated execution helps the compiler expose multiple paths of execution to the hardware. While the potential benefits for predicated execution are high, there are some costs involved. An instruction set must be designed that increases the number of source operands for all instructions in order to include the predicate operand.

Since all instructions in an architecture that supports full predication depend upon the predicate source operand, instructions must be added to the instruction set that efficiently modify the predicate registers. These instructions must be able to store, set, load, clear and define predicates.

2.6 Loop Transformations

Many classic code improving transformations that are performed by compilers as part of the optimization process involve loop transformations. Many of these loop transformations reduce loop overhead, which may reduce the number of branches executed during the run of a program. This paper discusses two of the most common loop optimizations, both of which help reduce the number of branches executed.

2.6.1 Loop Unrolling

Loop unrolling [8, 9] replicates the body of a loop n number of times, where n is the *loop unrolling* factor. The loop control code, must then be modified accordingly so that the loop behavior is preserved. Unrolling not only reduces the overhead of the loop, thus reducing the number of branches executed, but also provides new opportunities for other code improving transformations to be applied. However, loop unrolling can have a negative impact on the instruction cache since the unrolled loops are larger then the original loops.

2.6.2 Loop Unswitching

Loop unswitching [8, 10] is applied to loops that contain a branch with loop invariant conditions. The loop is replicated inside each fork of the branch, which saves the overhead of executing the comparison and branch on each iteration of the loop. This code improving transformation can reduce loop overhead and possibly enable parallelization.

2.7 Avoiding Conditional Branches

Other techniques use code duplication to try to remove conditional branches. These techniques try to find paths where the result of a branch is known and then duplicate code to take advantage of that fact. These code improving transformations can be viewed as a form of *partial redundancy elimination* for conditional branches.

2.7.1 Intraprocedural Conditional Branch Elimination

In the intraprocedural technique discussed in [11], the compiler must determine if branches can be avoided. To do this the compiler first calculates the set of registers and variables upon which the comparison associated with a branch depends. Next, the compiler attempts to determine if there exists a path from the point immediately after a conditional branch to the same path upon which the comparison is not affected. This is accomplished by calculating, for each block, an *in* and *out* state indicating the branches whose results are known at the beginning of the block and at the end of the block. A branch can become known at points within the control flow when a branch is executed, another branch subsumes the branch in question, or as an effect of some other instruction or side-effect in a block. Just as a branch can become known in a block, it can become unknown due to being affected by another instruction. A separate algorithm is used to duplicate paths where the state of the branch is known. In this case the branch can either be eliminated or converted to an unconditional jump.

2.7.2 Interprocedural Conditional Branch Elimination

The technique discussed in this section is very similar to the technique discussed in Section 2.7.1, and while both techniques eliminate conditional branches, the technique presented in this section works interprocedurally [12]. The compiler determines whether an interprocedural path exists along which a branch outcome is known at compile time, and then tries to eliminate it through code restructuring. If the path in question is affected by a procedure call, then the code restructuring involves *entry splitting* and *exit splitting*. These splitting techniques create multiple entries to a procedure and multiple exits from a procedure, respectively. The interprocedural conditional branch elimination technique presented here is demand driven, to avoid excessive computation and analysis of the program. Also the algorithm never increases the number of operations along a path through an interprocedural flow graph.

2.8 Transforming Conditional Branches into Indirect Jumps

Traditionally indirect jumps from tables are only generated when a compiler translates multiway select statements, such as the *C switch* statement. However, indirect jumps from tables can be used to replace sequences of branches which compare the same register or variable to some constants [13]. Creating efficient indirect jumps in the front end of a compiler is difficult. It is hard to know how instructions can be efficiently used in a machine independent environment, this is more effectively done after the machine instructions have been generated. Also opportunities for code improving transformations may be missed when indirect jumps are generated in the front end of a compiler. The technique discussed in this section advocates waiting until after code generation to create indirect jump tables as part of a general purpose transformation.

Performing an indirect jump from a table has been traditionally considered an expensive operation. It involves performing range checks, calculating the address of the table, calculating an offset into the table itself, loading the target from the jump table, and finally performing the jump itself. To make these jumps more efficient, several optimizations, such as padding the table and byte addressing may be applied.

2.9 Compiler Support for Predicated Execution

Predicated execution refers to the conditional execution of instructions based upon the value of a boolean source operand, called the *predicate.* The process of eliminating conditional branches from a program to use predicate support is called *if–conversion* [14, 15, 16]. However, using existing compiler support for architectures that support predicated execution has several problems. Typical uses of *if–conversion* combine all execution paths in a region into a single block. This forces instructions from the entire region to be examined *every* time a particular path is entered. The inclusion of infrequently executed paths and paths with a relatively large number of instructions can degrade the performance of the *if–converted* code. In addition, paths that have procedure calls or unresolvable memory accesses can also hinder optimizations phases and instruction selection.

2.9.1 The Hyperblock

A structure called the hyperblock [17] is used to overcome these problems. A hyperblock is a sequence of basic blocks in which control may enter only from the top (like basic blocks themselves). However, control may exit from one or more locations. Hyperblocks are similar to superblocks [18], but superblocks are not predicated and thus contain instructions from only one flow of control, while hyperblocks are predicated and may contain instructions from multiple flows of control. To extend a traditional compiler to more efficiently generate code for machines supporting predicated execution, we must select the basic blocks to be included in the hyperblocks, take the blocks selected and actually transform them into a hyperblock, and add some extensions to the compiler itself to be able to work with this new structure.

Not all basic blocks, to which control flows, are necessarily included within a hyperblock. Including all the basic blocks would combine all paths of execution, which could lead to a degradation of performance due to limited machine resources. Basic blocks are examined, with the goal of including them in a hyperblock, according to three criteria. The execution frequency of the block is considered because we want to exclude non-frequently executed blocks. Next, the size of the basic block is taken into account. Basic blocks of smaller size are given higher priority for inclusion as large blocks use more machine resources and may reduce performance. Finally, the characteristics of the instructions within the basic block are examined. Basic blocks that contain hazardous instructions, such as divides and unresolvable memory access are given less priority.

2.10 Control CPR for EPIC Architectures

EPIC architectures use three main features to facilitate compile time predication. Explicit parallel issue, speculation, and predication. To totally exploit EPIC processors compilers must transform and schedule application programs to make parallelism more available at run-time. The amount of parallelism in application programs is limited by both data dependencies and control dependencies.

Critical path reduction (CPR) is a set of techniques for transforming programs to reduce both data and control dependencies, thus enhancing parallelism. *Irredundant consecutive branch method* (ICBM) [19] is form of control CPR [20]. ICBM can greatly reduce the number of branches executed, which can improve performance. With traditional control flow techniques, operations are confined to basic blocks which are guarded by branches. Predication is used to guard operations without confining them to basic blocks, using *if-conversion*. Earlier work in if-conversion [21, 17] primarily eliminated branches within single-entry, single-exit program regions. Dependent chains of branches were not treated and remained dependent during program scheduling. With ICBM the program is partitioned into single entry acyclic regions and a predicate is associated with each region. The computation of these predicates and their use to guard blocks eliminate dependencies between non-speculative operations, which includes branches. These predicates are called *fully resolved predicates* (FRPs). The goal of ICBM is to reduce the critical path length by transforming code without increasing the average number of executed instructions, this is accomplished in part by moving branches off-trace.

2.11 TERA Computer System

The Tera architecture, a multistream MIMD system, was designed to be suitable for high speed implementations, to be applicable to a wide variety of problems, and for ease of compiler implementation. The first implementation of the architecture has 256 processors, 512 memory units, 256 input/output cache units, 256 input/output processors, 4096 interconnection network nodes and a clock cycle of 3 nanoseconds [22]. To help reduce branch costs, the Tera architecture has multiple versions of most operations, those that emit condition codes, and those that do not. When a branch occurs, it can examine a subset of the last 4 condition codes produced and branch accordingly. There are 8 target registers that are used as branch targets. The calculation of the branch target is separate from the decision to branch, allowing hardware to prefetch the instructions that calculate the branch target.

CHAPTER 3

CONDITION MERGING

3.1 Introduction

Conditional transfers of control occur frequently in programs, particularly in non-numerical applications. Conditional transfers of control are an impediment to improving performance since they consume a significant percentage of execution cycles, cause pipeline flushes when mispredicted, and can inhibit the application of other code-improving transformations. Techniques to reduce the number of executed branch instructions or remove branches from the control flow have the potential for significantly improving performance.

One approach to reducing the cost of conditional transfer of control is to attempt to merge a set of conditions together. Consider Figure 3.1(a), which shows conditions being tested in basic blocks 1, 3, and 5. The wider transitions between blocks shown in figures in this chapter represent the more frequently executed path, which occurs in Figure 3.1 when conditions a, b, and c are all satisfied. Figure 3.1(b) depicts the three conditions being merged together. If the merged condition is true, then the original conditions need not be tested. Note merging conditions results in the elimination of both comparison and branch instructions[1]. The elimination of the forks in the control flow between blocks 2, 4, and 6 may enable additional code-improving transformations to be performed. If the merged condition is not

[1]Blocks 2 and 3 in Figure 3.1(a) could have been represented as a single block. Throughout the chapter we represent basic blocks containing a condition as having no other instructions besides a comparison and a conditional branch so the examples may be more easily illustrated.

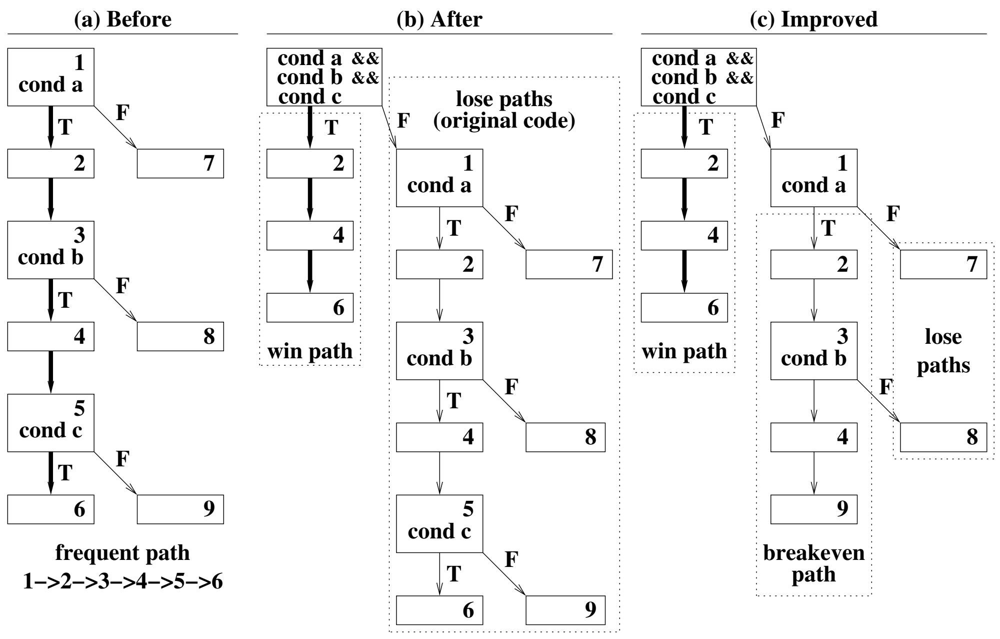


Figure 3.1: Merging Three Conditions

satisfied, then the original conditions are tested. Figure 3.1(c) shows that branches can become redundant after merging conditions. In this case, condition c must be false if (a && b && c) is false and (a && b) is true. Thus, the branch in block 5 can be replaced by an unconditional transition to block 9. We call this the *breakeven* path since the same number of branches will be executed in this path as were executed in the original path. We only apply the condition merging transformation when we estimate that the total instructions executed will be decreased.

In this chapter we describe techniques to replace the execution of a set of two or more branches with a single branch. Figure 3.2 presents an overview of the compilation process for merging conditions. A first compilation pass produces an executable file that is instrumented to collect path profiling information. A second compilation pass uses the profile data to merge conditions in the frequently executed paths. The C front-end used in this research is *lcc* [23] and the back-end used is VPO [24] targeted to the SPARC architecture. For our test programs these techniques on average decrease the number of branches by 15.81% (0.82% to 59.62%),

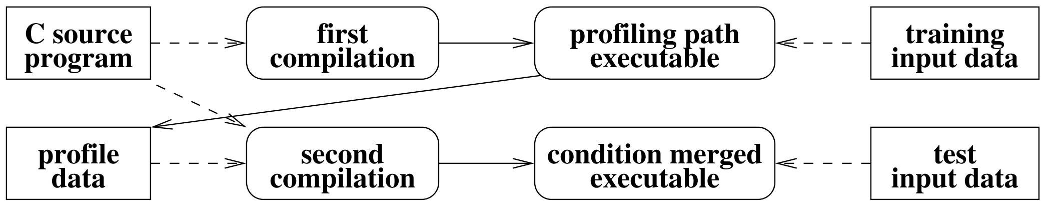


Figure 3.2: Compilation Process for Merging Conditions

the number of instructions by 5.74% (0.14% to 40.34%), and the execution time by 3.43% (-4.15% to 25.59%). These improvements were automatically obtained by the compiler on a conventional scalar processor.

3.2 Merging Conditions that Involve a Single Variable

In this section, we describe techniques to merge a set of conditions where each branch compares the same variable to invariant values (e.g., constants). In each case, the variable's value must not be unpredictably updated between the branches in the path.

3.2.1 Eliminating Logically Redundant Branches

Sometimes the conditions associated with two or more branches are logically correlated. In other words, one branch result (taken or fall through) may imply the result of another. Consider Figure 3.3(a). If the condition for the branch in block 1 is satisfied, then $v > 5$. If v is not affected between the execution of the two branches, then $v > 2$ is guaranteed to be satisfied and the branch in block 3 can be deleted, as shown in Figure 3.3(b). Other techniques using static analysis and code duplication could also eliminate the branch in block 3 [11, 12].

A path-based approach using profile data can be used to merge conditions that could not be merged using static analysis alone. Consider the flow graph in Figure 3.4(a). Satisfying the condition in block 1 will not guarantee the result of the branch in block 3. However, when the condition of the branch in block 3 is

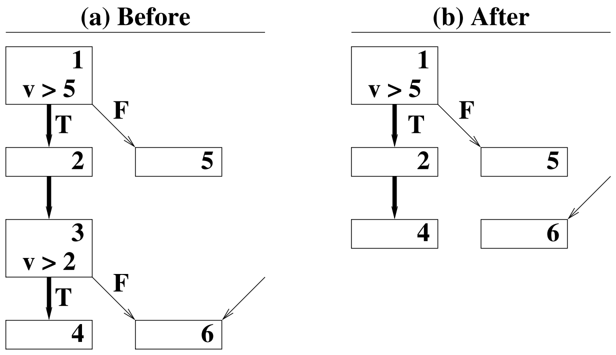


Figure 3.3: Merging Logically Redundant Conditions

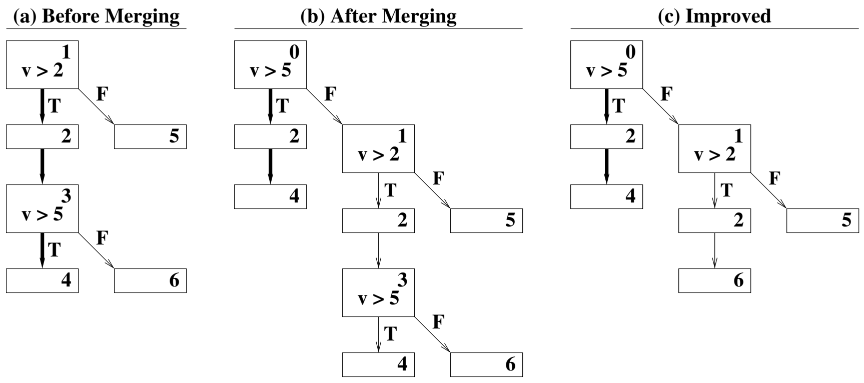


Figure 3.4: Merging Other Logically Redundant Conditions

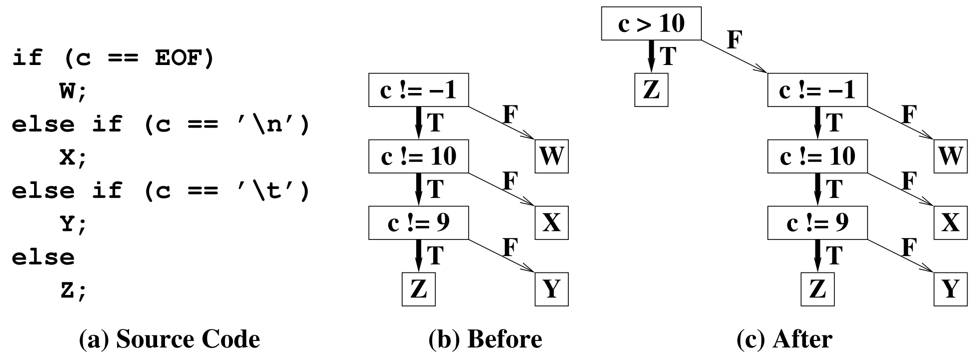
satisfied, then the condition in block 1 is guaranteed to be also satisfied. If the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ is the frequent path, then the second branch testing the condition $v > 5$ can be tested first, as shown in Figure 3.4(b). When the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ is taken, the condition of the branch in block 3 is guaranteed to be false since the same condition has already been shown to be false in block 0.

Dynamo, a dynamic optimization system [25], could also merge conditions like the ones depicted in Figure 3.4(a). However, in many cases, our technique is able to further improve the code by restructuring the flow graph. For example, the branch in block 3 is eliminated, as shown in Figure 3.4(c), since the test is now redundant.

Performing the code-improving transformation results in one less branch executed when path $0 \rightarrow 2 \rightarrow 4$ is taken, the same number of branches executed when the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 6$ is taken, and one additional branch executed when the path $0 \rightarrow 1 \rightarrow 5$ is taken. The actual improvement would depend on the frequency that each path is taken. However, a benefit can be obtained when the *win* path $0 \rightarrow 2 \rightarrow 4$, which reduces the number of branches, is taken more frequently than the *lose* path $0 \rightarrow 1 \rightarrow 5$, which increases the number of branches. Thus, improvements may be obtained even when the *break-even* path $0 \rightarrow 1 \rightarrow 2 \rightarrow 6$ is the most frequently executed path.

3.2.2 Merging Not Equal Tests Using Range Checks

A single variable is often checked to determine if it is equal to one of a set of constants. For example, Figure 3.5(a) shows the variable c being compared to three different character constants. Figure 3.5(b) depicts the flow graph representing these tests. It is often the case that a variable involved in such tests is not equal to any of the constants [26, 27]. Profile data is collected to determine not only the paths that are frequent, but also the reason that a particular transition was taken. For instance, when a variable is involved in a test to see if it is equal or not equal to a constant and the not equal transition is taken, then profile data is collected to determine if the variable was greater than or less than the constant. The compiler uses this profile information during the second compilation pass to determine the most likely range of values, which a particular variable may have, when it is not equal to any of the specified constants. If this range is greater than or less than all of the specified constants, then the set of branches can be replaced with a single branch. Figure 3.5(c) shows that the three branches testing the variable c can be bypassed by checking if c is larger than the largest specified constant. Note that when the merged condition is not satisfied, the original set of branches must be tested since the variable could still be equal to any of the specified constants. By performing condition merging on frequent paths, we can merge conditions separated by other intervening branches, which is not possible using a non-path based approach [26, 27].



```
if (c == EOF)
   W;
else if (c == '\n')
   X;
else if (c == '\t')
   Y;
else
   Z;
```

Figure 3.5: Merging Not Equal Tests Using Range Checks

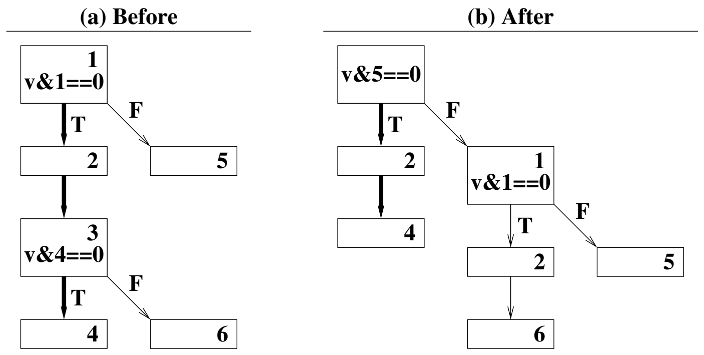


Figure 3.6: Merging Bit Tests

3.2.3 Merging Bit Tests

Different bits in a single variable are sometimes tested to see if they are clear or set. Figure 3.6(a) shows two different bits in the same variable being tested to see if they are clear. If the profile data indicates that one combination of bits is very likely, then that combination of bits can be tested in a single comparison by changing the constant being compared, as depicted in Figure 3.6(b). When the merged condition is not satisfied, both bits cannot be clear. If we reach block 1 from the block containing the merged condition, then we know that satisfying both conditions cannot occur since the merged condition failed. If we reach block 2 from block 1, then we know that the first condition is true. Thus, the second condition must be false and the branch in block 3 can be eliminated.

Table 3.1: SPARC Code Generation Strategies for Merged Bit Tests

General Bit Test	
`r[t]=r[v]&mask;`	# bitwise AND
`IC=r[t]?desired_value;`	# comparison
`PC=IC!=0,off_trace;`	# branch
Testing If Multiple Bits Are Clear	
`IC=(r[v]&mask})?0;`	# bitwise AND comparison
`PC=IC!=0,off_trace;`	# branch
Testing If Multiple Bits Are Set	
`r[t]=mask;`	# loop-invariant assignment
`IC=(r[t]&~r[v])?0;`	# bitwise ANDNOT comparison
`PC=IC!=0,off_trace;`	# branch

Table 3.1 shows different sequences of SPARC instructions represented as RTLs (register transfer lists) that can be used to test a set of bits. RTLs are machine and language-independent representations of machine specific instructions, used by many compilers as an intermediate language, including GCC and VPO (Very Portable Optimizer) [24], which is the compiler used to conduct this research. The first portion of the table shows a general sequence of three instructions that can be used to test if a set of bits has a specific combination of bits set. $r[v]$ is a register containing the value of the variable. $r[t]$ is a temporary. *Mask* is a constant that indicates the set of bits to be tested. *Desired_value* is the value of the bits that will result in the frequent path being taken. The branch is taken when the set of bits in the variable does not have the desired value. The second portion of the table shows that the sequence can be reduced to two instructions when the desired value is to have all of the bits clear. All integer ALU operations requiring a single cycle on the SPARC also have the option of performing a comparison to zero to set the integer condition codes. The third portion depicts the sequence of instructions we used when the desired value has all of the specified bits in the mask set. The SPARC has ANDNOT and ORNOT instructions that perform a bitwise NOT of the second operand before performing the logical operation. If the variable has all of the desired bits set, then the bitwise

NOT will cause all of the desired bits to be clear. Hence, the ANDNOT operation allows a comparison with zero. The first instruction assigning the mask value is loop invariant and the cost can be reduced by performing loop-invariant code motion if the branch is inside a loop and a register is available to hold the value of the mask.

Integer flags are commonly used in many applications. In effect, such variables are used as booleans, which requires only a single bit to be represented. Our system automatically reduces integer global and local scalar variables to bits within a global or local flags variable when such a scalar is only assigned constant values and is only dereferenced for '$= 0$' and '$\neq 0$' tests.

We accomplish the reduction of integer scalars to bits in a flag variable by examining and updating the *l*cc intermediate code files comprising the program. The assignment of nonzero constants to such a variable is replaced by a bitwise OR operation that sets a specific bit of a flag variable. The assignment of zero to such a variable is replaced by a bitwise AND operation that clears a specific bit of a flag variable. The '$= 0$' and '$\neq 0$' tests of the scalar variables are replaced by bitwise AND tests of the appropriate bit of the flag variable.

There are several advantages to reducing integer scalar variables to bits within a flag variable besides being able to merge conditions performing bit tests. Consider the SPARC instructions shown in Figure 3.7(a). Multiple assignments that set specific bits are merged by assigning the bitwise OR of the constants, as shown in Figure 3.7(b). Likewise, Figure 3.7(c) depicts that multiple assignments that clear specific bits are also merged. Redundant loads of the same flag variable are eliminated, as illustrated in Figure 3.7(d). Finally, Figure 3.7(e) shows that redundant stores are also eliminated. Note that the elimination of redundant loads and stores would not be possible if separate integer scalar variables were used. Performance conscious programmers often employ such techniques by hand, which is a tedious and error prone task. We are not aware of any prior work that automates this approach.

```
r[5] = M[flag];
r[5] = r[5] | 4;
r[5] = r[5] | 16;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~2;
r[5] = r[5] & ~8;
M[flag] = r[5];
```

(a) Original Instructions

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~2;
r[5] = r[5] & ~8;
M[flag] = r[5];
```

(b) After Merging Assignment Setting Bits

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(c) After Merging Assignment Clearing Bits

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(d) After Removing Redundant Loads

```
r[5] = M[flag];
r[5] = r[5] | 20;
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(e) After Eliminating Redundant Stores

Figure 3.7: Other Benefits from Reducing Scalars to Single Bits

3.3 Merging Conditions that Involve Multiple Variables

The previous section described techniques for merging sets of conditions that compare a single variable to invariant values. In this section we describe techniques for merging sets of conditions that involve multiple variables.

3.3.1 Merging into a Single Equivalent Condition

We use logical operations to efficiently merge conditions that compare multiple variables to 0 or -1 into a single equivalent condition. Consider the flow graph shown in Figure 3.8(a). The frequent path checks if the two variables are equal to zero. Figure 3.8(b) depicts the two conditions being merged together using a bitwise OR operation. Only when both variables are equal to zero will the result of the OR operation be zero. Again, if the merged condition is not satisfied, then testing the condition $v2 = 0$ is unnecessary since it cannot be true.

Unlike the methods presented in the previous section, the number of instructions required to merge conditions involving multiple variables is proportional to the number of different variables being compared. Figure 3.9 shows the general code generation strategy we used for merging such a set of conditions. $r[1] \ldots r[n]$ represent

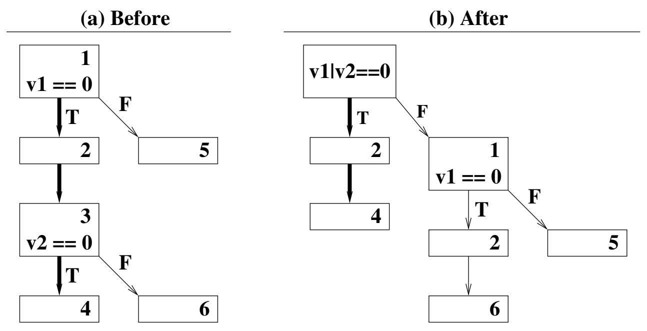


Figure 3.8: Merging Conditions That Test If Different Variables Are Equal to Zero

```
r[t] = r[1] | r[2];
r[t] = r[t] | r[3];
...
IC = (r[t] | r[n]) ? 0;
PC = IC != 0, <off-trace target>;
```

Figure 3.9: Code Generated for the Merged Condition That Checks If n Variables Are Equal to Zero

the registers containing the values of n different variables. $r[t]$ represents a register containing the temporary results. When merging conditions comparing n multiple variables, $n-1$ logical operations and a single branch replace $2n$ instructions (n pairs of comparisons and branches).

Table 3.2 shows how sets of conditions comparing multiple variables ($v_1 \ldots v_n$) to 0 and -1 can be merged into a single condition. Column one gives the rule number, column two depicts the original condition, column three depicts the merged condition, and column four represents the percentage of time the rule was selected during testing compared to the other rules shown in Table 3.2 and Table 3.3. Rule 1 has been illustrated in Figures 3.8 and 3.9. Rule 2 uses the SPARC ORNOT instruction to perform a bitwise NOT on an operand before performing a bitwise OR operation. A word containing the value -1 has all of its bits set in a two's complement representation. Thus, if the operand is a -1, then the result of the bitwise NOT would be 0 and at that point the first rule can be used. The merged

Table 3.2: Rules for Merging Conditions Comparing Multiple Variables into a Single Equivalent Condition

Rule	Original Conditions	Merged Condition	% Applied
1	$v_1 = 0$ && $\cdots$ && $v_n = 0$	$(v_1 \mid \cdots \mid v_n) = 0$	42.7%
2	$v_1 = 0$ && $\cdots$ && $v_n =$ -1	$(v_1 \mid \cdots \mid \sim v_n) = 0$	0.0%
3	$v_1 < 0$ && $\cdots$ && $v_n < 0$	$(v_1$ & $\cdots$ & $v_n) < 0$	0.0%
4	$v_1 \geq 0$ && $\cdots$ && $v_n \geq 0$	$(v_1 \mid \cdots \mid v_n) \geq 0$	4.5%
5	$v_1 < 0$ && $\cdots$ && $v_n \geq 0$	$(v_1$ & $\cdots$ & $\sim v_n) < 0$	0.9%
6	$v_1 \geq 0$ && $\cdots$ && $v_n < 0$	$(v_1 \mid \cdots \mid \sim v_n) \geq 0$	0.0%

condition in rule 3 performs a bitwise AND operation on the variables. A negative value in two's complement representation has its most significant bit set. Only if the most significant bit is set in all of the variables will the most significant bit be set in the result of the bitwise AND operation. If the most significant bit is set in the result, then the result value will be negative. The merged condition in rule 4 performs a bitwise OR operation on the variables. A nonnegative value in a two's complement representation has its most significant bit clear. Only if the most significant bit is clear in all the variables will the most significant bit be clear in the result of the bitwise OR operation. The last two rules perform a bitwise NOT on an operand, which flips the most significant bit along with the other bits in the value. This allows '<' and '>=' tests to be merged together.

Notice that '> 0' and '<= 0' tests are not listed in the table. A '> 0' test would have to determine that both the most significant bit is clear and that one or more of the other bits are set. These types of tests cannot be efficiently performed using a single logical operation on a conventional scalar processor.

Additional opportunities for condition merging become available when sets of conditions, which cross loop boundaries, are considered. It would appear that in Figure 3.10(a) there is no opportunity for merging conditions. However, Figure 3.10(b) depicts that after loop unrolling there are multiple branches that use the same relational operator. Our system merges sets of conditions across loop iterations.

```
for (i = 0; i < 10000; i++)
   if (a[i] < 0)
      x;
```

(a) Original Code

```
for (i = 0; i < 10000; i += 2) {
   if (a[i] < 0)
      x;
   if (a[i+1] < 0)
      x;
}
```

(b) After Loop Unrolling

Figure 3.10: Increasing Merging Opportunities by Unrolling Loops

In order to simplify the analysis, we only merge conditions that span two consecutive iterations of a loop.

3.3.2 Merging into a Sufficient Condition

Our system also uses logical operations to efficiently merge conditions, which compare multiple variables, into a single sufficient condition. In other words, the success of the merged condition will imply the success of the original conditions. However, the failure of the merged condition does not imply the original conditions were false. Table 3.3 shows rules for merging conditions containing multiple variables into a single sufficient condition. The columns in this table contain similar information to the columns in Table 3.2.

There were no '$\neq 0$' tests listed in Table 3.2. Yet tests to determine if a variable is *not-equal* to zero occur frequently in programs. We can determine if two or more variables are guaranteed to be not equal to zero by performing a bitwise AND operation on the variables and checking if the result does not equal to zero, as shown in rule 7 of Table 3.3. Note that failure of the merged condition does not imply that the variables are all not equal to zero.

One may ask how often such conditions can be successfully merged in practice. Consider the code segment:

```
if (p1 != NULL && p2 != NULL)
```

Table 3.3: Rules for Merging Conditions Comparing Multiple Variables into a Single Sufficient Condition

Rule	Original Conditions	Merged Condition	% Applied
7	$v_1 \neq 0$ && $\cdots$ && $v_n \neq 0$	$(v_1 \,\&\cdots\&\, v_n) \neq 0$	18.2%
8	$v_1 \neq c_1$ && $\cdots$ && $v_n \neq c_n$	$(v_1 \,\&\cdots\&\, v_n) \;\&\; \sim(c_1 \mid\cdots\mid c_n) \neq 0$	15.5%
9	$v_1 \neq c_1$ && $\cdots$ && $v_n \neq c_n$	$\sim(v_1 \mid\cdots\mid v_n) \;\&\; (c_1 \mid\cdots\mid c_n) \neq 0$	16.4%
10	$v_1 < c_1$ && $\cdots$ && $v_n < c_n$	$(v_1 \mid\cdots\mid v_n)\; u< \;\min(c_1, \cdots, c_n)$	1.8%

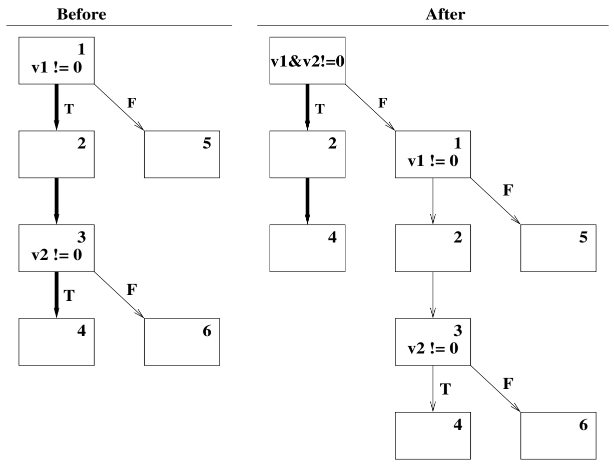


Figure 3.11: Merging Conditions That Check If Different Variables Are Not Equal to Zero

where two pointer variables, $p1$ and $p2$, are tested to see if they are both non-NULL. In most applications, a pointer variable is only NULL in an exceptional case (e.g., end of a linked list). It is extremely likely that two non-NULL pointer values will have one or more corresponding bits both set due to address locality. Figure 3.11 shows how two or more conditions checking if multiple variables are not equal to zero can be merged. If the merged condition is not satisfied, then the entire original set of branches still needs to be tested.

We are also able to merge conditions that check if multiple variables are all not equal to a specified list of constants. One method we used is to check if any bits set in all of the variables are always clear in all of the constants. Rule 8 of Table 3.3 depicts how this is accomplished, where $c_1 \ldots c_n$ are constants. A bitwise AND

$$\begin{aligned}(v_1\& \cdots \& v_n)\& \sim (c_1| \cdots |c_n) &\neq 0\\ (v_1\& \cdots \& v_n)\& \sim (0| \cdots |0) &\neq 0\\ (v_1\& \cdots \& v_n)\& \sim (0) &\neq 0\\ (v_1\& \cdots \& v_n)\& 0xFFFFFFFF &\neq 0\\ (v_1\& \cdots \& v_n) &\neq 0\end{aligned}$$

Figure 3.12: Rule 7 is Implied by Rule 8

operation is performed on all of the variables to determine which bits are set in all of these variables. The complement of the bitwise OR of the constants is taken, which results in the bits being set that are clear in all of the constants. Note that determining which bits in the constants are always clear is determined at compile time. If any bits set in all of the variables are clear in all of the constants, then it is known that all of the variables will not be equal to all of the constants. Rule 8 in Table 3.3 can also be used when the constants are all zero. Figure 3.12 shows how the merged condition in rule 8 simplifies to the merged condition in rule 7 when $c_1 \ldots c_n$ are all zero.

Another method to determine if multiple variables are not equal to a list of constants is to check if one or more bits, which are clear in all of the variables, are set in all of the constants. Merging conditions using this method is shown in rule 9 of Table 3.3. The decision to use rule 8 or 9 is determined by checking the success of these rules during the profile run. Given an equal likelihood that either rule could be successfully applied, rule 8 is preferable since rule 9 requires an extra instruction to perform the bitwise NOT operation, which is accomplished at compile time for rule 8.

We found that rules 8 or 9 can be used when a single variable was checked to see if it was not equal to a set of constants. For instance, Figure 3.13 illustrates how rule 8 is simplified to only require a single bitwise AND operation to set the condition codes when only a single variable was involved. Thus, both the range checking method illustrated in Figure 3.5 and the bit testing methods using rules 8 or 9 are checked to estimate which would be most profitable.

$$\begin{aligned}(v_1\& \cdots \& v_1)\& \sim (c_1| \cdots |c_n) &\neq 0\\ v_1\& \sim (c_1| \cdots |c_n) &\neq 0\end{aligned}$$

Figure 3.13: Using Rule 8 Efficiently with a Single Variable

We are also able to merge conditions checking if multiple variables are less than (or less than or equal to) constants. Rule 10 in Table 3.3 depicts how this is accomplished, where $c_1 \ldots c_n$ must be positive constants and the $u<$ in the merged condition represents an unsigned less than comparison. If the result of the bitwise OR on the variables is less than the minimum constant, then the original conditions have to be satisfied. The unsigned less than operator is necessary since one of the variables could be negative and the result of the bitwise OR operation would be treated as a negative value if a signed less than operation is used.

Our system merges conditions comparing multiple variables to values that are not constants. Consider the original loop and unrolled loop shown in Figure 3.14(a) and Figure 3.14(b), respectively. It would appear there is no opportunity for merging conditions. However, x is loop invariant. Thus, bits that are set in both $a[i]$ and $a[i+1]$ can be ANDed with $\sim x$ to determine if the array elements are not equal to x, as shown in Figure 3.14(c). The expression $\sim x$ is loop invariant and the compiler will move it out of the loop when loop-invariant code motion is performed. Likewise, the loads of the two array elements in the *else* case will be eliminated after applying common subexpression elimination. Note we are also able to merge conditions checking if multiple variables are not equal to multiple loop invariant values. The profitability of using rules 8 and 9, where $c_1 \ldots c_n$ could be loop invariant values, is estimated during the profile run.

Figure 3.15 shows another example where conditions comparing multiple unsigned variables to non-constants can be merged. When finding the largest value in an array, it is very likely that most of the elements examined will not be greater than the maximum value found so far. When merging the less than or equal tests, the value (*max* in this case) being compared to the variables needs to be the same value

```
for (i=0; i < 10000; i++)
   if (a[i] == x)
      num++;
```

(a) Original Code

```
for (i=0; i < 10000; i += 2){
   if (a[i] == x)
      num++;
   if (a[i+1] == x);
      num++;
}
```

(b) After Loop Unrolling

```
for (i=0; i < 10000; i +=2){
   if ((a[i] & a[i+1]) & ~x)
      continue;
   else{
      if (a[i] == x)
         num++;
      if (a[i+1] == x)
         num++;
   }
}
```

(c) After Condition Merging

Figure 3.14: Merging Conditions That Check If Multiple Variables Are Not Equal to Loop-Invariant Values

(otherwise one would not know which would be the minimum) and has to be invariant in the path between the conditions.

3.4 Estimating the Benefit of Merging a Set of Conditions

In order to apply a condition merging transformation, the compiler needs to know which paths are frequently executed in the program. We extended the *EASE* environment [7] in the *VPO* compiler [24] to collect path profile information. We define a path as a sequence of blocks within a function that are either terminated by edges that cross loop boundaries or when a return block is reached. Consider the example flow graph that is shown in Figure 3.16. The edges that cross loop boundaries are $3 \rightarrow 4$ (entering the loop), $6 \rightarrow 4$ (backedge), and $6 \rightarrow 7$ (exiting the loop). Statically enumerating all of the paths in a function according to this definition can sometimes result in an excessive number of paths. Thus, we decided to detect paths dynamically during the execution of the profile run. During the first compilation pass we modify the generated assembly to insert a call to an instrumentation routine at the beginning of each basic block. When this routine

```
max = 0;
for (1 = 0; i < 10000; i++)
   if (a[i] > max)
      max = a[i];
```

(a) Original Code

```
max = 0;
for (i = 0; i < 10000; i += 2)
   {
   if (a[i] > max)
      max = a[i];
   if (a[i+1] > max)
      max = a[i+1];
}
```

(b) After Loop Unrolling

```
max = 0;
for (i = 0; i < 10000; i + = 2){
   if ((a[i] | a[i+1]) <= max)
      continue;
   else{
      if (a[i] > max)
         max = a[i];
      if (a[i+1] > max)
         max = a[i+1];
   }
}
```

(c) After Condition Merging

Figure 3.15: Merging Conditions That Check If Multiple Unsigned Variables Are Less Than an Unsigned Invariant Value

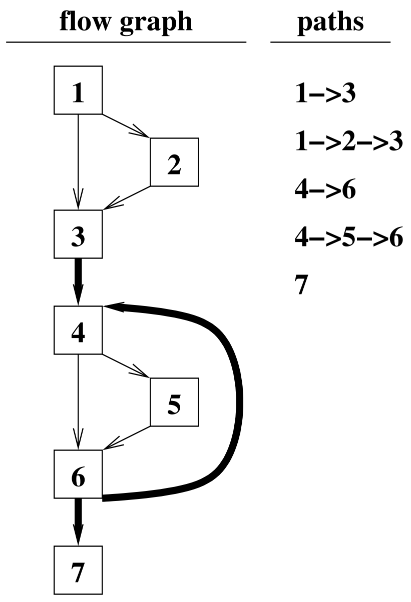


Figure 3.16: Paths That Do Not Cross Loop Boundaries

is invoked, the current block number is appended to the list of blocks that have been encountered for the current path. When we detect a block that could not be in the same path and it is the first time the path is encountered, we record the list of blocks as a path and we increment a counter for that path. We found that many functions had a large number of static paths, but often relatively few of these paths were actually executed. Whenever the dynamic number of unique paths in a function exceeded a specified limit (100 in our experiments), we stopped collecting measurements and no longer attempted to merge sets of conditions for that function. When this limit was exceeded, which rarely occurred in our test programs, we felt that it was unlikely to find a frequent path within such a function since so many paths are executed.

We also attempted to merge sets of conditions across the backedges of loops. We only attempted to merge conditions that span two consecutive executions of the same path by treating it as a single path. We believe this restriction allows most of the beneficial sets of conditions across path boundaries to be merged without significantly increasing the number of sets of conditions to be evaluated.

We use the path profile information for merging conditions in the following manner. We only examine paths whose execution contributed to more than 0.1% of the total instructions executed in a program. Consider the flow graph in Figure 3.17. Assume that the conditions in blocks 2 and 6 can be merged together, where both branch conditions are assumed to be satisfied. This example illustrates two interesting points. First, a path starting at block 2 and ending at block 6 comprises only a subpath of any path represented in the path profile information. Second, there are two possible subpaths ($2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ and $2 \rightarrow 4 \rightarrow 6$) connecting these two branches even when the condition in the first branch is satisfied. Rather than duplicating entire paths when merging sets of conditions, we instead duplicated subpaths. The compiler collects all possible subpaths that contain the specified set of conditions. It then uses the path frequency information to estimate the benefit of

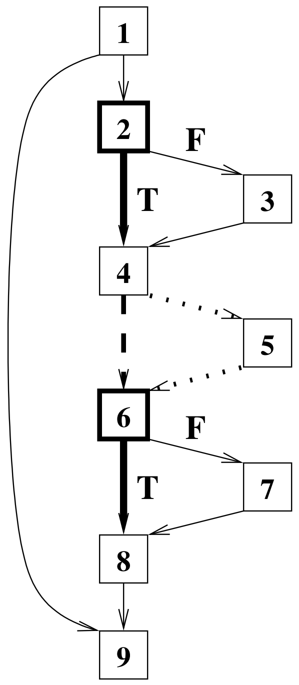


Figure 3.17: A Set of Conditions to be Merged May be Connected by Multiple Subpaths

merging each set. The compiler determines the likelihood that the path containing these branches with the appropriate transitions will be taken given that the first branch in the set will be reached. The benefit is estimated based on the instructions saved when these conditions are satisfied and the extra instructions executed when the conditions are not satisfied.

3.5 Applying the Transformation

Figure 3.18 gives a high-level overview of the algorithm used to merge sets of conditions. After finding all of the sets of conditions, which can be merged, the compiler sorts these sets in descending order, according to estimated benefit. There are two reasons for merging the most beneficial sets first:

1. Merging may require the generation of loop-invariant expressions that can be moved out of the loop after applying loop-invariant code motion. This

transformation requires the allocation of registers for which there are only a limited number available on a target machine.

2. Merging conditions changes the control flow within a function. If two sets of conditions overlap in the paths of blocks that connect them, then the estimated benefit is invalid after the first set is merged and the second set of conditions will not be merged.

```
for each path executing > 0.1% of the total insts {
    for each branch in the path {
        for each of the remaining branches in the path {
            Determine if the set of branches can be merged.
        }
    }
    for each mergeable set of conditions in the path {
        Estimate the benefit of sets based on the expected gain.
    }
}
Sort the sets in descending order of benefit.
for each set with a benefit {
    if set blocks are not affected by a previously merged set {
        if the set has the registers needed to apply the merging transformation {
            Merge the set of conditions.
            Mark blocks that are affected.
        }
    }
}
Reapply other code improving transformations.
```

Figure 3.18: Overview of Condition Merging Optimization

After merging sets of conditions, we apply a number of code-improving transformations in an attempt to improve the modified control flow. For instance, Figure 3.1(c) shows that merging conditions simplifies the control flow in the *win* and *breakeven* paths. Our general strategy was to generate the control flow in a simple manner and rely on other code-improving transformations to improve the code. For instance, the code shown in Figures 3.14(c) and 3.15(c) can be improved by applying loop-invariant code motion and common subexpression elimination. We also invoke a number of code-improving control-flow transformations, such as branch chaining, code positioning, loop inversion, and unreachable code elimination, to improve the code layout. Branch chaining simplifies the control flow when the target block of a transfer of control contains a single unconditional jump, which may also have a target block containing a single unconditional jump. The first transfer of control is re-written to jump to the last target in the chain of unconditional jumps. Code positioning can reorder the blocks in the control flow. For example if the target of an unconditional jump does not have a fall through predecessor, then the target block, along with any contiguous blocks that follow it, can be placed directly after the block with the unconditional jump. The unconditional jump can then be removed. Loop inversion places a loop exit test at the end of the loop instead of the beginning of the loop, saving the execution of an unconditional jump in the loop.

3.6 Results

Table 3.4 shows the test programs on which condition merging was applied. We chose these non-numerical applications since they have complex control flow, a higher density of conditional branches, and have branches testing integer values. Control dependences in such applications often inhibit many types of compiler optimizations. For each SPEC benchmark we used training and test data that were available with the benchmark. For the other applications we used input data similar to the examples found in the man pages describing these applications. In each case the training data was smaller than the test data, resulting in fewer instructions executed in the training run than in the test runs reported in this section.

Table 3.4: Test Programs

SPEC Benchmarks	Description
compress	Compresses and decompresses files.
go	AI game playing program.
ijpeg	Graphic compression and decompression.
li	LISP interpreter.
m88ksim	Motorola 88K simulator.
perl	Practical extraction and report language.
vortex	Database program.
Other Applications	**Description**
ctags	Generates a tag file for vi.
dd	Copies a file with possible conversions.
deroff	Removes *nroff* constructs from a file.
diff	Displays line-by-line differences between two text files.
grep	Searches files for a string or regular expression.
lex	Lexical analysis program generator.
nroff	Text formatter.
od	Dumps files in a variety of formats.
othello	Game playing program.
pr	Prepares file(s) for printing.
sed	Stream editor.
sort	Sorts and collates lines.
tbl	Formats tables for *nroff*.
tr	Substitutes or deletes selected characters in text.
uniq	Report or filter out repeated lines in a file.
yacc	Yet another compiler-compiler.

For those rules that involved merging multiple variables, the majority of the rules applied were for inequality to the value zero or a set of constants. The percentage that each multiple variable rule was selected is shown in column four of Table 3.2 and Table 3.3.

Table 3.5 shows the overall impact that each condition merging techniques had on the number of branches performed and the total number of instructions executed. The baseline measurements included the reduction of local and global scalar variables to bits within flag variables, which achieved slight reductions in the number of instructions executed as illustrated in Figure 3.7. Techniques *A* and *B* eliminate conditions that could not be eliminated using non-path based approaches. For instance, Figure 3.4 shows an example of eliminating a logically redundant condition

Table 3.5: Dynamic Results from Applying the Individual Techniques

Label	Description	Branches	Insts.
A	merging using logical correlation	95.899%	98.539%
B	merging using range checks	93.537%	96.431%
C	merging using bit tests	98.008%	99.284%
D	merging using logical operations	88.462%	96.066%
A-B	applying techniques A and B	89.961%	95.556%
A-C	applying techniques A, B, and C	88.279%	95.089%
A-D	applying techniques A, B, C, and D	84.189%	94.256%

using technique *A* that could not be eliminated using static analysis alone [11]. Technique *B* eliminates conditions that are separated by other intervening branches, which is also not feasible using a non-path based approach [26, 27]. Note that techniques *A*, *B*, and *C* merge sets of conditions that compare a single variable and technique *D* merges sets of conditions that can compare multiple variables. However, technique *C*, which merges bit tests, relies on the reduction of different scalar variables to bits within flag variables. Thus, technique *C* in effect merges conditions that compare multiple variables specified in the source code. In fact, many of the sets of conditions merged using technique *C* would have been merged using technique *D* if reducing scalars to bits had not been applied. Table 3.5 also shows techniques *C* and *D* reduced the number of branches by 6.42% over techniques *A* and *B*. In fact, technique *D* alone reduced more branches than techniques *A* and *B* combined. Overall, 15.81% of the executed branches were eliminated.

The primary goal of our study was to reduce the number of conditional transfers of control executed. However, it is interesting to note the effect on other performance measures, even though we did not tune the compiler to exploit the modified control flow. Table 3.5 also shows the effect that condition merging had on the number of instructions executed. Overall, there was a 5.74% average reduction in the number of instructions executed. While most of the reduction was directly due to fewer executed comparisons and branches, occasionally the compiler was able to apply

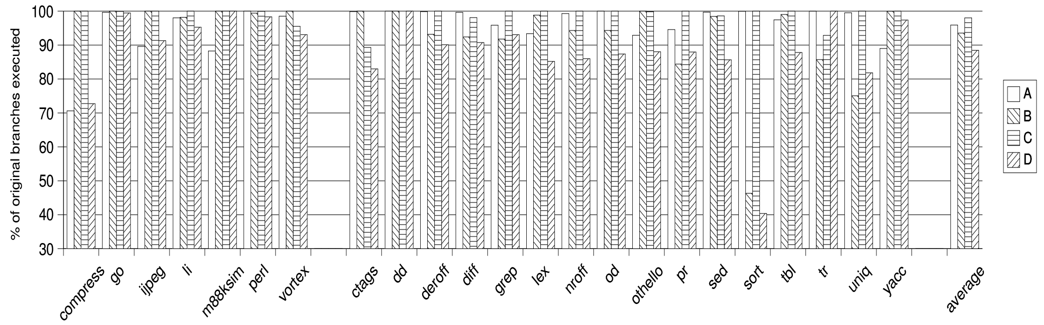


Figure 3.19: Effect on Branches Executed for Each Technique

code-improving transformations on the modified control flow to obtain additional improvements.

Figures 3.19–3.21 display condition merging results for each program. Figure 3.19 displays the effects of each individual technique on the number of branches. While using logical operations (technique D) and range checks (technique B) were the most beneficial, branches were merged by applying rules from all of the techniques. Figures 3.20 and 3.21 show the cumulative results of applying the techniques. *Sort* had unusually large benefits since most of the execution was spent in tight loops where conditions could be merged using techniques B and D.

In a few cases, the application of an additional technique increased the number of branches and/or instructions executed. This increase was due in part to using different training and test data, which affected the accuracy of our estimated benefits. The number of instructions executed increased more frequently since we could not predict the effect that the modified control flow would have on subsequently applied optimizations.

There are reasons why the cumulative benefits were less than the sum of the benefits from the individual techniques. (1) Sets of conditions merged by one technique may also be merged by using a different technique. For instance, sets of $\neq$ conditions in *sort* could be merged using logical operations (technique D) or by using

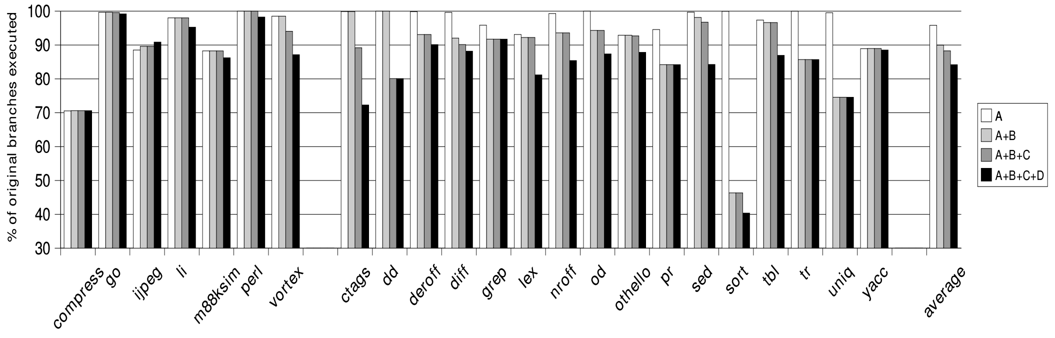


Figure 3.20: Cumulative Effect on Branches Executed

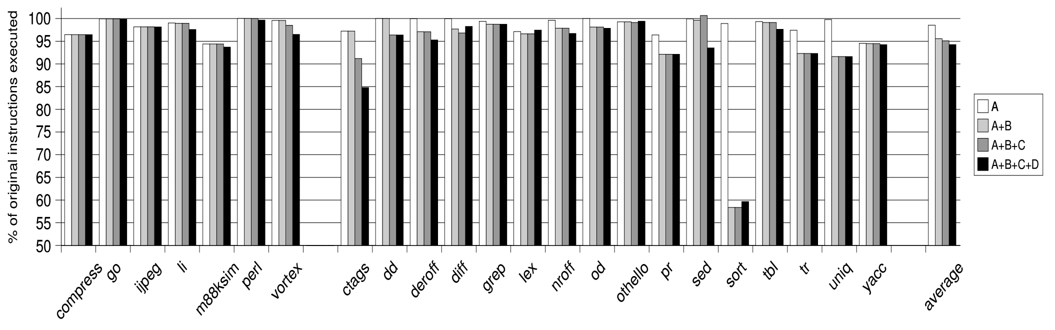


Figure 3.21: Cumulative Effect on Instructions Executed

range checks (technique B). (2) Sometimes different sets of conditions to be merged overlapped in the control-flow graph. When two sets of conditions overlap and it was estimated that both could be merged separately with benefits, we only merged the set deemed most beneficial. In this case the merging of the first set of conditions will change the paths associated with the second set of branches. (3) Merging conditions may require the allocation of registers for loop-invariant values. Merging one set of branches sometimes consumed the remaining available registers that are needed to merge another set.

Figure 3.22 shows the effect that condition merging had on execution time for each of the test programs. For the UltraSPARC II, we found that there was a high

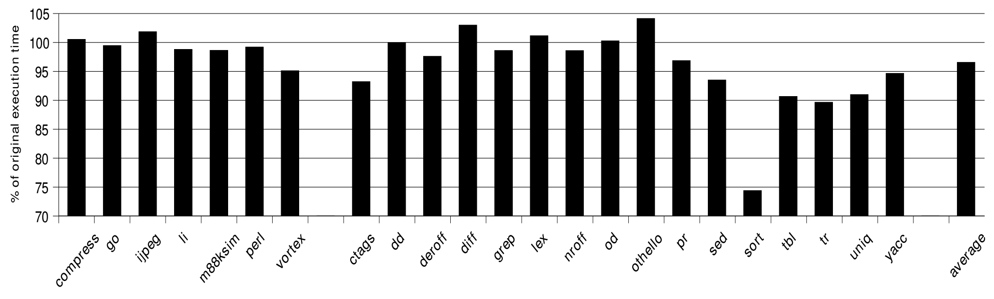


Figure 3.22: Effect on Execution Time

variance in the *user* time using the Unix utilities available (e.g., *ptime*). Thus, we instead measured execution time by reading the 64-bit clock register (%*tick*) that is accessible on the UltraSPARC architectures. We read the value in this clock register before and after each external call to a run-time library routine, which allowed us to obtain for each program a fairly consistent execution time that does not include the time spent in the run-time library. We used the minimum time of 200 executions of each program, which represents the execution time with the fewest cycles spent in other processes due to interrupts. Figure 3.22 shows that on average that there was a 3.43% execution time reduction on the UltraSPARC II. While the execution time of most of the applications improved, there was a performance degradation for a couple of the applications. The reason for the degradation is difficult to explain fully given the complexity of the UltraSPARC II implementation.

Applying all the techniques resulted in a roughly 7% increase in static code size after condition merging. This static increase compares favorably with the 5.74% decrease in instructions executed. The code size would have increased more if we had not required that the paths we inspected comprise at least 0.1% of the total instructions executed.

Figure 3.23 shows the effect that condition merging had on instruction cache performance for a 16KB two-way set associative cache with 32 byte lines, which is

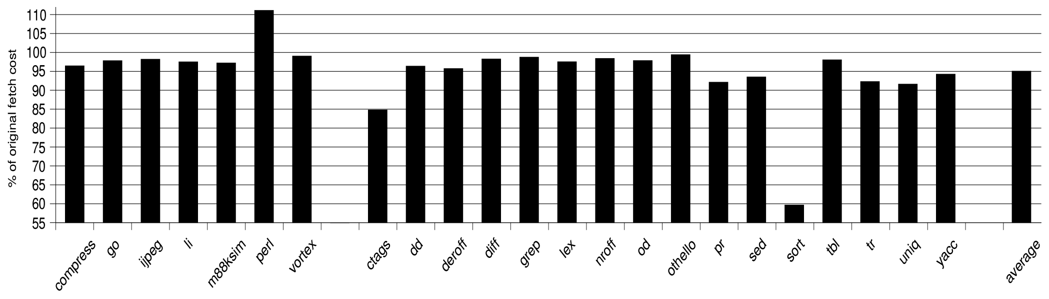


Figure 3.23: Effect on Estimated Fetch Cost for a 16KB Two-Way Associative Instruction Cache

the configuration used on the UltraSPARC II. We found that the average miss ratio increased slightly (+0.00092) after merging conditions. However, the miss ratio is not an appropriate measurement given that the number of instructions executed was affected. In the figure we show the effect on the overall fetch cost, which we calculated by counting each hit as a single cycle and each miss as ten cycles [28]. The average fetch cost decreased by almost 5%. As expected, there is a very strong correlation between instructions executed (A+B+C+D result in Figure 3.21) and fetch cost (Figure 3.23) since the instruction cache hit ratios were high. The only program whose fetch cost increased was *perl*. A frequently executed loop, which involved code from multiple routines, was aligned slightly differently after merging conditions and this alignment caused the performance to degrade since the routines were then mapped to the same cache lines. After merging, *perl* actually had slightly fewer instructions executed and its code size only increased by about 0.1%. Figure 3.24 shows the effect condition merging had on the fetch cost for a variety of cache sizes, where each configuration was two-way associative with a 32 byte line size. In each case the average fetch cost was improved.

On many machines there is a pipeline stall associated with every taken branch. Merging conditions resulted in approximately 28% average reduction in the number

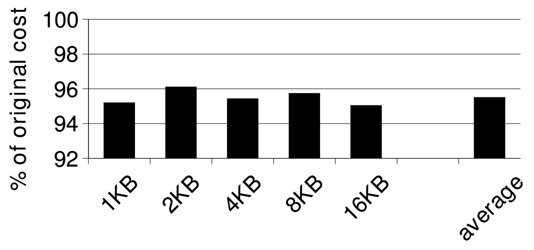


Figure 3.24: Average Fetch Cost for a Variety of Cache Sizes

of taken branches for the test programs. This reduction was due in part to decreasing the number of executed branches. The transformation also makes branches within the *win* path more likely to fall through since the sequence of blocks representing each frequent path is now contiguous in memory. Overall, the average number of control transfers (taken branches, unconditional jumps, calls, returns, and indirect jumps) was reduced by roughly 20%.

On average there were only 10.13 sets of conditions merged per test program. Some sets, while beneficial, were not merged due to overlapping regions of code. However, the main reason that some beneficial sets are not merged is infrequent execution. Table 3.6 shows the number conditions, per set, that were merged. Due to these sets of conditions being frequently executed, significant control height reduction is achieved despite only merging on average 10.13 sets of conditions per test program and 2.29 conditions per set.

Table 3.6: The Number of Conditions Merged.

Conditions Merged	Percentage of Sets
2	82.83%
3	9.01%
4	6.01%
5	0.86%
6	1.29%

CHAPTER 4

REDUCING COSTS BY DECOUPLING THE SPECIFICATION OF THE COMPARISON

As previously discussed, a conditional transfer of control can be broken into three distinct parts: a comparison portion, the calculation of the branch target address and the actual transfer of control. This chapter presents two techniques, both of which decouple the definition of the values that are involved in a comparison from the comparison itself. Both techniques try to remove explicit comparison instructions from the instruction stream, in an attempt to reach the branch instruction sooner.

A conditional transfer of control is traditionally implemented using two separate instructions: a comparison and a branch. The comparison instruction usually sets a register (condition code, predicate, or general-purpose) whose value will be accessed by the branch instruction to determine the flow of control to follow. The branch target address is normally encoded within the branch instruction itself, often as a displacement. The branch instruction is also used to indicate the point at which the transfer of control should occur.

The advantage of using the traditional approach is that the transfer of control can be easily encoded, as there is a separate instruction for the actual comparison. The disadvantage is that two instructions have to be fetched and decoded, and the program may stall at the comparison instruction if the operands are not ready. In contrast, execution is not typically stalled on most processors when a branch instruction is reached. Instead a branch prediction is made and the next *predicted* instruction is fetched and executed. However, when a comparison instruction is stalled on an in-order processor, the following branch instruction, where the prediction is made, is also delayed.

A conditional transfer of control can also be implemented using a single instruction that performs both the comparison and branch. The advantage of this technique is that there is only one instruction so the branch is often reached sooner and thus the branch prediction is made sooner. However, it is difficult to encode the information necessary to represent all of the types of comparisons and the branch offset in the bits available for a single instruction. To encode this information within a single instruction, typically fewer bits are allocated to specify the branch target address limiting the range of branches. Likewise, comparisons to an arbitrary constant cannot be made.

In both of the new techniques described in this chapter a new instruction called a *comparison specification* (*cmpspec*) is used to define the values that will be used in a comparison, when it is reached. However, the cmpspec only specifies what values will be used in the comparison. It does not actually perform the comparison. Therefore, it does not have any dependencies with the instructions that produce the values. Unlike the technique discussed in Chapter 3, these two techniques require slight hardware modifications, which are smaller and simpler than hardware typically used for branch prediction. The hardware needed for the first technique is slightly more complex than the hardware needed for the second technique.

The first technique presented in this chapter uses cmpspecs to specify the values, as well as the relational operator used in the comparison. The values involved in a comparison can either be a register or a constant. A new comparison register file is used to hold an index into the general-purpose register file or into a constant table if the comparison value is a constant. A comparison register specifies the two values to be used in the comparison, one of which must be a register. When a cmpspec is executed, implicit comparisons for the first register specified are enabled. When an assignment to a register (for which implicit comparisons are enabled) is encountered, then a comparison between the two values specified occurs and the result is stored in a branch register, that will be accessed by a branch instruction to determine the outcome of the branch.

The second technique presented in this chapter also decouples the comparison definition with the actual comparison itself. This technique developed from the implicit comparisons in an attempt to minimize certain problems inherent with implicit comparisons, including hardware complexity. This technique does not have *implicit comparisons*. Instead, we have a single instruction that performs both the comparison and branch. Unlike similar techniques that use a single compare and branch instruction, not all the information needed for the comparison is encoded within the instruction that performs the comparison and branch since much of it is encoded in the cmpspec. Using this approach we avoid the disadvantages of other common techniques that use a single comparison and branch instruction.

Using separate instructions to perform the comparison and the branch is typically how conditional transfers of control are implemented. Our techniques attempt to isolate invariant portions of the work involved with conditional transfers of control and expose them to other compiler optimizations, such as loop-invariant code motion and common subexpression elimination. These optimizations can often detect opportunities for code improvement on the exposed portion of a conditional transfer of control in cases where it is not possible with traditional comparison instructions. In many cases, execution time was reduced because useful work is performed where a comparison might stall using traditional compare and branch instructions.

While these two approaches could be applied to dynamically scheduled machines, it is more likely that a traditional comparison instruction would be scheduled in parallel with other instructions, reducing the benefits that might be obtained from using cmpspecs. However, this work is ideally suited for many statically scheduled embedded systems, as we are removing many comparison instructions from the critical path. We evaluate a number of benchmark programs and present results showing reductions in the number of instructions executed, as well as cycle counts to test the effectiveness of these techniques.

4.1 Implicit Comparisons

For this technique, cmpspecs are generated that define the comparison that should occur. Each time a register is assigned a value and implicit comparisons have been enabled for that register, a comparison is performed between the value being assigned to the register and a second specified value. This approach eliminates the need, in many cases, for the execution of an explicit comparison instruction and allows the branch to be accessed and predicted earlier in the instruction stream. However, several simple modifications, such as new types of registers and a compare unit, are needed to facilitate implicit comparisons.

4.1.1 Exploiting Implicit Comparisons

In this section we provide a description of implicit comparisons and several examples of how code is generated using these new instructions. If we cannot reduce the number of instructions needed to perform a comparison and branch, then explicit comparisons in conjunction with branch instructions will be used. For this research, implicit comparisons are only enabled for general-purpose registers, not floating-point registers.

Figure 4.1 depicts a high level algorithm used when generating code containing implicit comparisons. The following subsections illustrate how the algorithm works. The algorithm, which is applied to the generated code late in the compilation process, begins by searching for comparisons in loops within a function, in order of the most deeply nested loops first. We then check to see if this comparison could be legally generated in the implicit manner. Using an implicit comparison may not be possible if a reserved register is involved in the comparison (e.g., the register being assigned a value is a parameter to a function.) Next, we determine if there are multiple comparisons involving the same registers within one live range. If so, these comparison instructions are stored in a *1 of M* list of comparisons to be dealt with later. Next, we determine if registers involved in the comparison would need to be renamed and if there are any available registers. The next step is to determine if there exists a path from the preheader (where the cmpspec will be inserted) to the branch, along which there is *not* an assignment that will cause the implicit comparison to occur. If such a path exists, then we insert a move that will force an implicit comparison in the preheader.

```
for all loops beginning with the most deeply nested {
    for all comparison instructions {
        if implicit comparison legal {
            if 1 of M conflict {
                Add candidate to list of M candidates.
                continue
            }
            if renaming required && a register is available {
                rename required registers.
            } else {
                continue
            }
            if forced implicit comparison required && legal {
                insert move instruction into preheader.
            }
            generate cmpspec in loop preheader.
        }
    }
    for each separate 1 of M list {
        Select 1 comparison to make implicit.
    }
}
```

Figure 4.1: High Level Algorithm for Implicit Comparisons

If we have passed all these checks, then an implicit comparison is generated. Comparison specifications must precede all the implicit comparisons that reference them in order for the information to be available when a comparison is performed as they are placed in the preheader of the current loop. When there are no more comparison instructions in a loop, we generate comparisons from those previously inserted into the *1 of M* lists. For each loop, there will be a separate *1 of M* list for each set of comparisons that are within a single live range. The remainder of this section contains code examples written in pseudo-code to represent source level code and RTLs to represent machine instructions.

4.1.1.1 Basic Implicit Comparisons

Figure 4.2 shows a section of code comprising a *for* loop. The body of the loop is not shown as it is not relevant to the example. Figure 4.3(a) shows the code represented as RTLs with an *explicit* comparison. The *b* registers are branch registers that contain a bit indicating if the branch condition is true.

```
for (i = 0; i < 100; i++)
    ...
```

Figure 4.2: C Code

Figure 4.3(b) shows the RTLs using an implicit comparison instruction instead of a normal explicit comparison. Whenever a comparison specification is executed, implicit comparisons for the register referenced in the instruction are enabled. During the execution of the function, every assignment to a register specified in a cmpspec, be it a load or an arithmetic operation, will result in a corresponding implicit

```
    r[9]=0;
  L3:
    ...
    r[9]=r[9]+1;
    b[9]=r[9]<100;
    PC=b[9],L3;
```

(a) Explicit Comparison

```
    r[9]=0;
  L3:
    ...
    c[9]=#100; o[9]='<';
    r[9]=r[9]+1;b[9]=r[9]+1<#100;
    PC=b[9],L3;
```

(b) Implicit Comparison

Figure 4.3: Loop With Different Types of Comparisons

comparison. Implicit comparison effects, shown in bold italics in the examples throughout this chapter, would not be explicitly encoded within instructions, but are included in the examples to illustrate when useful implicit comparisons occur.

A cmpspec has been inserted in line 4 of Figure 4.3(b). The effect `c[9]=#100;` indicates that whenever register `r[9]` is assigned a new value, that new value will be compared with the constant 100. Note that the constant will be sign extended and stored in the constant table. The same index used for the comparison register (9) will also be used to access the constant in the constant table. The effect `o[9]='<';` indicates the type of comparison that will be performed every time register `r[9]` is assigned a value. The result from the implicit comparison in line 5 is stored in branch register `b[9]`, which is then accessed on line 6 to determine if the branch should be taken.

Although the code in Figure 4.3(b) has been modified to use implicit comparisons, it was generated in a *naive* manner. That is, the code is checked for legality and generated, but we rely upon standard optimizations, such as loop-invariant code motion and common subexpression elimination, to further optimize the code.

4.1.1.2 Implicit Comparisons Checks

Before a comparison can be made implicit there are several checks the compiler must make. For a comparison involving the values stored in two general-purpose registers the assignments to one must post-dominate all the assignments to the other in the live range of the two registers. Figure 4.4 shows a control-flow graph where either an assignment to `r[8]` in block B or to register `r[21]` in block C can be performed last before reaching the explicit comparison. In this case, the compiler cannot make the comparison implicit and still preserve the semantics of the function.

Other problems that may inhibit the formation of implicit comparisons are running out of available registers (when renaming is required), comparisons occurring outside of loops, conflicts with other comparisons in a single live range of a register

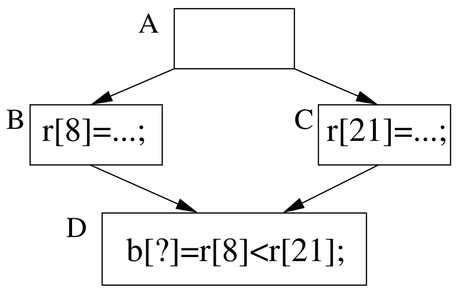


Figure 4.4: Post-Domination Requirement

and the predefined use of a register (e.g., calling conventions) in a comparison when renaming is necessary. All these remaining problems are discussed in more detail in the following sections.

4.1.1.3 Using Loop-Invariant Code Motion

When generating implicit comparisons, we must insert a comparison specification containing the details of the implicit comparisons that occur when values are assigned to a specific register. When a branch becomes implicit (i.e., it depends upon an implicit comparison), the explicit comparison is removed at the cost of adding another instruction, a comparison specification. To make implicit comparisons beneficial, we need to amortize the cost of the cmpspec.

The code in Figure 4.3(b) is equivalent in cost with the traditional method of performing branches shown in Figure 4.3(a), with regard to the number of instructions executed. We have eliminated the explicit comparison, but added the cmpspec on line 4. Comparison specifications can usually be moved out of a loop because the value being assigned to the comparison register is either a register index, or a constant (which are both loop invariant). There can only be only one cmpspec, for a specific register, in a loop since the scope of the specification is the entire loop.

Figure 4.5 depicts the code from Figure 4.3(b) after the cmpspec has been moved to line 2 in the loop preheader. Now the cost for the cmpspec is only incurred before the loop is entered, not on every iteration.

```
    r[9]=0;
    c[9]=#100; o[9]='<';
L3:
    ...
    r[9]=r[9]+1;b[9]=r[9]+1<#100;
    PC=b[9],L3;
```

Figure 4.5: After Loop Invariant Code Motion

```
 1     ...
 2     c[8]=16;o[8]='<';
 3     r[8]=#val;b[8]=#val<r[16];
 4     PC=b[8],L3;
 5     ...
 6 L3:
 7     c[8]=16;o[8]='<';
 8     r[8]=#val2;b[8]=#val2<r[16];
 9     PC=b[8],L5;
10 L5:
11     ...
```

(a) Before CSE

```
 1     ...
 2     c[8]=16; o[8]='<';
 3     r[8]=#val;b[8]=#val<r[16];
 4     PC=b[8],L3;
 5     ...
 6 L3:
 7     r[8]=#val2;b[8]=#val2<r[16];
 8     PC=b[8],L5;
 9 L5:
10     ...
```

(b) After CSE

Figure 4.6: Applying CSE to Implicit Comparisons

If the cmpspec for an implicit comparison cannot be removed from the loop, then we have gained nothing. For this reason, only explicit comparisons that occur within loops are candidates for becoming implicit. The code for branches outside of loops are generated in the traditional manner using explicit comparisons.

4.1.1.4 Using Common Subexpression Elimination

Common subexpression elimination (CSE) is a code improving transformation that can eliminate instructions that compute values that are already available. Figure 4.6(a) shows a section of code with two implicit comparisons. The cmpspec in line 7 is the same as the comparison specification on line 2. Since the specification on line 7 is dominated by the one in line 2, it will be eliminated by CSE, as illustrated in Figure 4.6(b). This would not be possible if explicit comparisons were used since the actual comparison of the values still has to occur.

4.1.1.5 Using Comparison Negation

To allow CSE to eliminate more comparison specifications, the types of boolean tests that can be specified in a cmpspec have been limited to those illustrated in column one of Table 4.1. Comparisons involving tests not defined in column one of the table, will be constructed using the tests in column one and a negation of the results. The equivalent negation of these tests are depicted in column two.

Table 4.1: Test Definitions and their Complements

Tests Used	Complementary Test
==	!=
<	>=
>	<=

Figure 4.7(a) shows a section of code involving two implicit comparisons. When the RTLs are generated, normally the comparison conditions are reversed, for better layout of the basic blocks. However, in this example the test definition portion of the cmpspec on line 8 is not reversed even though the opposite condition needs to be checked. To preserve the semantics of the branch on line 10, the branch occurs on the *negation* (!) of the comparison result derived from the implicit comparison on line 9. At this point the comparisons specifications on lines 2 and 8 are identical and line 8 will be removed by CSE, as depicted in Figure 4.7(b).

When determining if a cmpspec can be moved into the preheader of a loop, renaming of the registers in the cmpspec as well as the implicit comparison and branch instructions may be necessary. If there are several cmpspecs defining an implicit comparison for the same register in a loop that are not within the same live range, then renaming *may* allow all of the cmpspecs to be moved outside the loop. Figure 4.8(a) shows a loop in which register `r[8]` is used for two separate comparisons in lines 3 and 9. The corresponding cmpspecs are not within the same live range for register `r[8]`. However, we cannot move both comparisons

```
 1     r[9]=#val1;
 2     c[8]=9; o[8]='<';
 3     r[8]=#val2;b[8]=#val2<r[9];
 4     PC=b[8],L1;
 5     ...
 6 L1:
 7     ...
 8     c[8]=9; o[8]='<';
 9     r[8]=#val3;b[8]=#val3<r[9];
10     PC=!b[8],L2;
11     ...
12 L2:
13     ...
```

(a) Negating the Branch Register

```
 1     r[9]=#val1;
 2     c[8]=9; o[8]='<';
 3     r[8]=#val2;b[8]=#val2<r[9];
 4     PC=b[8],L1;
 5     ...
 6 L1:
 7     ...
 8     r[8]=#val3;b[8]=#val3<r[9];
 9     PC=!b[8],L2;
10     ...
11 L2:
12     ...
```

(b) After CSE

Figure 4.7: Comparison Specification with Branch Register Negation

```
 1 L2:
 2     c[8]=12; o[8]='<';
 3     r[8]=r[5]+2;b[8]=r[5]+2<r[12];
 4     PC=b[8],L3;
 5     ...
 6 L3:
 7     ...
 8     c[8]=17; o[8]='==';
 9     r[8]=MEM;b[8]=MEM==r[17];
10     PC=b[8],L4;
11     r[11]=r[8]+7;
12 L4:
13     ...
14     /* Loop to L2, or Fallthru */
15     ...
```

(a) Conflicting Comparison Specifications

```
 1 L2:
 2     c[8]=12; o[8]='<';
 3     r[8]=r[5]+2;b[8]=r[5]+2<r[12];
 4     PC=b[8],L3;
 5     ...
 6 L3:
 7     ...
 8     c[9]=17; o[9]='==';
 9     r[9]=MEM;b[9]=MEM==r[17];
10     PC=b[9],L4;
11     r[11]=r[9]+7;
12 L4:
13     ...
14     /* Loop to L2, or Fallthru */
15     ...
```

(b) Rename Register `r[8]` to `r[9]`

Figure 4.8: Register Renaming in a Single Live Range

specifications (in lines 2 and 8) into the loop preheader because each cmpspec defines a different implicit comparison for register `r[8]`.

4.1.1.6 Using Register Renaming

The solution is to rename register `r[8]` in one comparison specification and its corresponding instructions, such as the assignment and branch instruction. Figure 4.8(b) shows the code after register `r[8]` in the second cmpspec in line 8 has been renamed to register `r[9]`. The assignment that initiates the implicit comparisons in line 9 was changed to use register `r[9]`. The branch instruction

```
1      c[8]=12; o[8]='<';
2      c[9]=17; o[9]='=';
3  L2:
4      r[8]=r[5]+2;b[8]=r[5]+2<r[12];
5      PC=b[8],L3;
6      ...
7  L3:
8      ...
9      r[9]=MEM;b[9]=MEM==r[17];
10     PC=b[9],L4;
11     r[11]=r[9]+7;
12 L4:
13     ...
14     /* Loop to L2, or Fallthru */
15     ...
```

Figure 4.9: Both Comparison Specifications Moved Outside the Loop

in line 10 now references branch register `b[9]`. The use of register `r[8]` in line 11 also had to be renamed to register `r[9]` as well since it is part of the live range. After renaming the register, the two cmpspec instructions in lines 2 and 8, no longer conflict and are both moved outside of the loop, as shown in Figure 4.9.

4.1.1.7 Choosing 1 of M Comparisons

If two or more cmpspecs referring to the same register exist in a loop and fall within a single live range for that register, then only one of the cmpspecs may be moved out of the loop. This situation can occur if we are performing multiple tests on a single variable, each test being for a different condition. To solve this dilemma, all the comparisons that reference the same register in the same live range are gathered into a list. Out of the M comparisons in the list we can only choose one to make implicit. The rest of the conditional transfers of control are generated using explicit comparisons.

Figure 4.10(a) depicts a case where we have two comparison specifications that reference the same register in lines 3 and 11. The cmpspec instructions define different comparison parameters for an assignment to register `r[17]`. Both the assignments

```
 1 L1:
 2    r[16]=#val;
 3    c[17]=16;o[17]='<';
 4    r[17]=#val2;b[17]=#val2<r[16];
 5    PC=b[17],L3;
 6    ...
 7    r[12]=r[1]+r[17];
 8    PC=L5;
 9 L3:
10    ...
11    c[17]=8;o[17]='==';
12    r[17]=#val3;b[17]=#val3==r[8];
13    PC=b[17],L5;
14    ...
15    /* Loop to L1 or fallthru */
16 L5:
17    ...
18    r[8]=r[17]+r[5];
```

(a) Choosing Branches

```
 1    ...
 2    c[17]=16;o[17]='<';
 3 L1:
 4    r[16]=#val;
 5    r[17]=#val2;b[17]=#val2<r[16];
 6    PC=b[17],L3;
 7    ...
 8    r[12]=r[1]+r[17];
 9    PC=L5;
10 L3:
11    ...
12    r[17]=#val3;
13    b[17]=r[17]==r[8];
14    PC=b[17],L5;
15    ...
16    /* Loop to L1 or fallthru */
17 L5:
18    ...
19    r[8]=r[17]+r[5];
```

(b) Making Candidate Invariant

Figure 4.10: Making 1 of M Candidates Invariant

in lines 4 and 12 can reach the use of register `r[17]` in line 18. Thus, all three instructions are part of one live range for register `r[17]`. If we move both cmpspecs out of the loop, the code would be illegal.

The solution is to choose 1 of the `M` candidate comparisons to make implicit. The comparison chosen has its cmpspec generated and moved into the loop preheader while the remaining comparison(s) will be generated in the explicit format, as illustrated in Figure 4.10(b). The first branch now has an implicit comparison and the second remains explicit.

4.1.1.8 Inserting Forced Implicit Comparisons

For a cmpspec to be beneficial, it must be moved into the loop preheader. However, it is possible that there is some path from the loop preheader to the basic block containing the branch, where the register associated with the implicit comparison (and specified by the cmpspec) is not assigned a value.

Figure 4.11(a) shows a loop, where the comparison specifications have already been moved into the preheader of the loop. The implicit comparison specified by the cmpspec in line 2 is fine, however the implicit comparison defined in line 3 has a

```
1     /*PREHEADER*/
2     c[9]=8; o[9]='=';
3     c[10]=11; o[10]='>';
4     /*BLOCK A*/
5  L4:
6     r[8]=#val1;
7     r[9]=#val2;b[9]=#val2=r[8];
8     PC=b[9],L1;
9     /*BLOCK B*/
10    ...
11    PC=L2;
12    /*BLOCK C*/
13 L1:
14    r[10]=#val3;b[10]=#val3>r[11];
15    ...
16    /*BLOCK D*/
17 L2:   ...
18    PC=b[10],L4;
```

(a) Path to Branch Without an Implicit Comparison

```
1     /*PREHEADER*/
2     c[9]=8; o[9]='=';
3     c[10]=11; o[10]='>';
4     r[10]=r[10];b[10]=r[10]>r[11];
5     /*BLOCK A*/
6  L4:
7     r[8]=#val1;
8     r[9]=#val2;b[9]=#val2=r[8];
9     PC=b[9],L1;
10    /*BLOCK B*/
11    ...
12    PC=L2;
13    /*BLOCK C*/
14 L1:
15    r[10]=#val3;b[10]=#val3>r[11];
16    ...
17    /*BLOCK D*/
18 L2: ...
19    PC=b[10],L4;
```

(b) Loop with a Forced Move

Figure 4.11: Adding a Forced Move to a Loop

problem. If the path through this loop is blocks $A \rightarrow C \rightarrow D$ everything works as intended. But if the path through the loop is blocks $A \rightarrow B \rightarrow D$ then an implicit comparison to set branch register `b[10]` never occurs.

Figure 4.11(b) shows the solution to the problem. A *move* is inserted into the preheader after the cmpspec. When the preheader is entered, the implicit comparison in line 4, will execute before the branch in line 19 is reached. Although the move is an extra instruction, the cost is only incurred once, before the loop is entered, not on every subsequent iteration.

4.1.2 New Hardware

Before defining the hardware required for this implementation of implicit comparisons, we must define the assumptions for our baseline architecture. We are assuming a statically scheduled, single issue machine that is common for many low power embedded processors. The baseline architecture contains single bit branch registers, which are set by comparison instructions and read by branch instructions. The architecture also contains register windows.

Opcode	C Reg	Cmp	f	Constant or Index
31–24	23–19	18–16	15	14–0

Figure 4.12: Comparison Specification Encoding

To implement implicit comparisons, the values to be compared must be specified. Comparison specifications are a new instruction that requires a new instruction format. A proposed encoding is illustrated in Figure 4.12. We assume that eight bits are needed for the opcode itself (31-24), which would allow for a total of 256 different instructions. Five bits are needed to encode the comparison register (23-19). Three bits are needed to encode all the types of comparisons allowed (18-16). One bit (15) is reserved as a flag used to indicate if the value stored in bits 14-0 is an index or a constant value. If the flag is 1, then the 5 least significant bits (4-0) specify an index into the general-purpose register file. If the flag is 0, then bits (14-0) comprise a constant, that will be stored in the constant table, at the index indicated by the five bits that compose the comparison register's index (23-19). It is unnecessary to specify which branch register to use in an implicit comparison since there is a one-to-one correspondence between the comparison registers and the branch registers. The implicit comparison itself is not explicitly encoded, since it is performed in hardware whenever a specified register is assigned a value.

Figure 4.13 depicts a simplified version of a classical five-stage pipeline showing the hardware additions needed for implicit comparisons in bold. For implicit comparisons to be implemented, a few new architectural features need to be added. A new register file (for comparison and operational registers), a constant table, new ports for the conventional register file, a new compare unit (CU), and their interconnections will need to be added. A third register read port has been added to the general-purpose register file to access the general-purpose register that will sometimes be used in implicit comparisons. The comparison register file is accessed when a general-purpose register is set and implicit comparisons are enabled for that

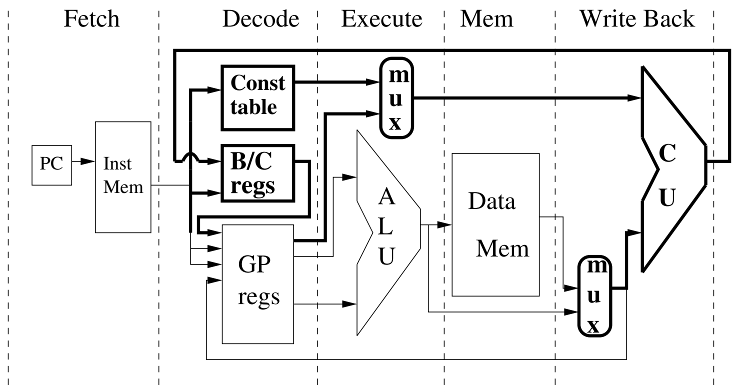


Figure 4.13: Five-Stage Pipeline with Additional Hardware

register. Implicit comparisons involve two values. The first comparison value is the one assigned to the register being set. An index for the second comparison value is stored in a comparison register. This index can indicate which general-purpose register holds the second comparison value or where in the constant table the second comparison value is stored. The constant table is a separate table that holds sign-extended values of constants specified to be used in an implicit comparison. Also stored in the register file are the operations, which indicates the type of comparison that will be performed. Throughout this chapter comparison and operational registers are depicted as separate registers, but they can be viewed as one register since all the information for the comparison and operational registers requires only 9 bits and the registers are **always** referenced in pairs.

The information indicating when and how implicit comparisons are performed is stored in the comparison register. For these experiments we are assuming register windows for both the general-purpose and comparison registers. Implicit comparisons are disabled, by default when a function is entered. Implicit comparisons are enabled, for a specific register, when a comparison specification instruction is executed. A cmpspec that defines the implicit comparison for register `r[8]` turns on implicit comparisons only for instructions that set register `r[8]`. Implicit comparisons are also automatically re-enabled after returning from a function due to accessing the comparison registers in the previous register window.

The implicit comparison, which is performed by the CU, is depicted as taking place in the *write back* stage in Figure 4.13. One input to the CU is the *new* value of the register that is being set. The new value of the register being compared is available in this stage, whether it comes from a load or an arithmetic instruction. Alternatively we could have additional CUs in other stages, and use them to perform the comparison when the inputs are available earlier. The second comparison value is specified by a comparison register and accessed in the *decode* stage. The other input to the CU is either a constant from the constant table or the value of a general-purpose register. Forwarding hardware is needed to forward the value of the general-purpose register that was specified in case the value has changed since it was accessed in the decode stage. The result of the implicit comparison is written to a corresponding branch register. Because the branch register only holds the outcome of comparisons, they each can be represented with a single bit, i.e., 32 bits is all that is needed to represent all of the branch registers.

A drawback to delaying the comparison is that the branch misprediction penalty is increased. Although the misprediction penalty for mispredicted implicit branches can be longer than for normal branches, the results presented in Section 4.1.3 show that the decrease in number of instructions executed outweighs the increased misprediction penalty for almost all of the programs tested.

4.1.3 Results

Our approach of using implicit comparisons was tested using a large subset of benchmarks from the MiBench suite of benchmarks [29], described in Table 4.2. We could not get several of the benchmarks from this suite to compile on our system due to problems with the benchmark specifications. The benchmarks were compiled with the VPO compiler, which is integrated with EASE (Environment for Architecture

Study and Experimentation) [30]. Using EASE we were able to simulate a machine that contained branch registers, comparison registers, operational registers and a constant table.

Table 4.2: Benchmarks Tested

Name	Description
adpcm	adaptive pulse code modulation encoder
basicmath	simple math calculations
bitcount	bit manipulations
blowfish	block encryption
crc32	cyclic redundancy check
dijkstra	shortest path problem
fft	fast Fourier transform
ijpeg	image compression
ispell	spell checker
lame	MP3 encoder
patricia	routing using reduced trees
qsort	quick sort of strings
rsynth	text-to-speech analysis
sha	exchange of cryptographic keys
stringsearch	search words
susan	image recognition
tiff	convert a color TIFF image to b/w image

Code was generated using the new implicit comparison instructions and dynamic instruction counts and cycles times were obtained. Figure 4.14 shows a chart illustrating the percentage improvement for the number of instructions executed, which was on average 5%. The improvements ranged from 0.01% (tiff) to almost 18% (stringsearch). Roughly 60% of comparisons on average became implicit using this method. The average dynamic percentage of instructions that were explicit comparisons in the benchmarks originally was 12.7%. This percentage can be viewed as an upper bound on the number of instructions that can be reduced by eliminating explicit comparisons.

Table 4.3 shows the different reasons why a comparison might not be generated in the implicit manner, and the total percentage of comparisons that fell into that category. Case 1 is where the comparisons did not occur within a loop. Case 2 indicates when register renaming was necessary and there were no registers available to move the cmpspecs outside of the loop. Case 3 represents when there were M different comparisons of the same register in the same live range, which means that only one could be selected. Finally, case 4 is where a comparison could not be generated in the implicit manner because it involves the predefined use of a register (e.g., a register is a parameter to a function call that could not be renamed).

Table 4.3: Reasons Comparisons Could not be Made Implicit

Case	Percentage	Reason
1	39%	Non-Loop
2	35%	No registers available
3	13%	M comparisons in a live range
4	13%	Predefined use of a register

Execution cycles were calculated by estimating the cycles each basic block would take to execute, given assumptions about the operational latencies for different types of instructions. These operational latencies, shown in Table 4.4, were used by other researchers in a prior study [19]. We also assumed redundant CUs were available to avoid structural hazards due to instructions with different latencies. The reduction in instructions executed and the reduction in cycles shown in Figures 4.14 and 4.15 respectively are highly correlated.

Branch registers may be set by either explicit or implicit comparisons. Branches that depend on implicit comparisons have a higher misprediction penalty, when the implicit comparison occurs close to the actual branch instruction since the implicit comparison occurs later in the pipeline. A correlating branch predictor was used to perform branch prediction. When a branch misprediction was encountered, it was determined how many cycles earlier the comparison occurred and the appropriate penalty was added to the count of executed cycles. The misprediction ratio for the benchmarks tested, was typically unaffected when using implicit comparisons.

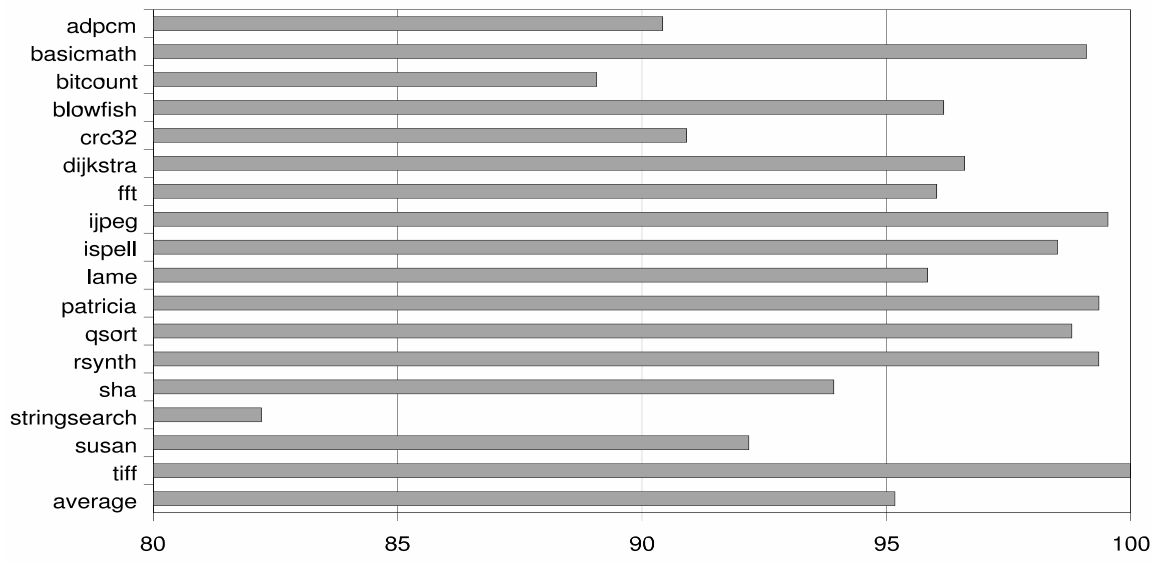


Figure 4.14: Percentage of Instructions Executed

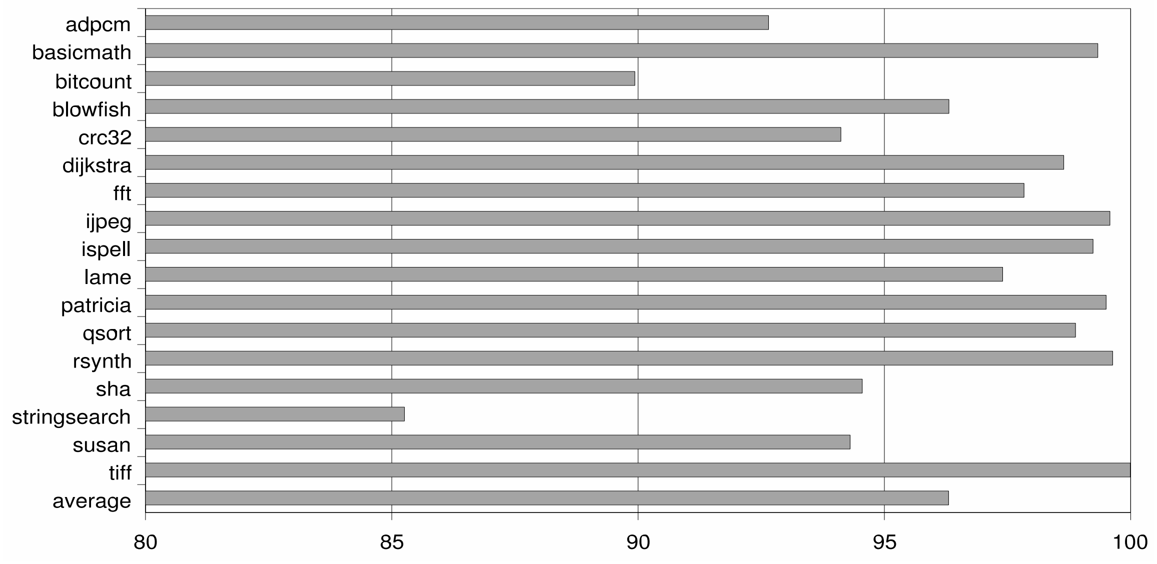


Figure 4.15: Percentage of Execution Cycles

Table 4.4: Operational Latencies

Operation	Latency
Simple Integer	1
Floating Point	3
Memory Load	2
Memory Store	1
Integer and Floating Point Multiply	3
Integer and Floating Point Divide	8
Branch	1

Figure 4.15 shows the percentage reduction in number of cycles each benchmark took to complete successfully, which was on average 4%. The improvements ranged from 0.01% (tiff) to almost 15% (stringsearch). The reduction in cycles needed to execute a program is not as great as the reduction in instructions executed for two reasons. First, we have a greater misprediction penalty for mispredicted branches that depend on implicit comparisons. Second, multiple cycles are sometimes needed to execute each instruction. The number of cycles required to execute each block depends on several factors including the instruction types and dependencies with other instructions that may cause stalls.

4.2 Comparison Specifications with Cbranches

The technique for reducing the cost of traditional transfers of control presented in this section also involves decoupling the values to be compared with the actual comparison like the technique discussed in the previous section. However, instead of implicit comparisons, this technique generates a single compare and branch instruction (*cbranch*) along with a comparison specification (*cmpspec*) instruction. This technique retains the advantages from other techniques that use a single compare and branch instruction without incurring the disadvantages. Cmpspecs define the values that are to be compared, but the comparison does not occur until the compare and branch instruction is reached. Similar to implicit comparisons, the branch instruction is usually reached sooner than the approaches that use separate comparison and branch instructions. Because cmpspecs only reference register numbers and not register values, other compiler optimizations, such as loop-invariant code motion and common subexpression elimination, can often be applied to cmpspecs, in cases where it would not be possible with traditional comparison instructions. Because there is less information encoded within our cbranch instructions, our technique does not significantly decrease the branch target address range or limit the values that can be compared. To test the effectiveness of our technique, we evaluated a number of benchmarks from the MiBench suite of programs on an ARM processor and present results showing average reductions of 5.6% instructions performed and 5.2% cycles executed.

4.2.1 New Hardware

The underlying architecture used for these experiments is the ARM processor [31]. We chose the ARM since this is a popular embedded ISA for which there exists a commonly used simulator, SimpleScalar [32]. The ARM also has separate comparison and branch instructions and thus was a good fit for this technique. The baseline ARM

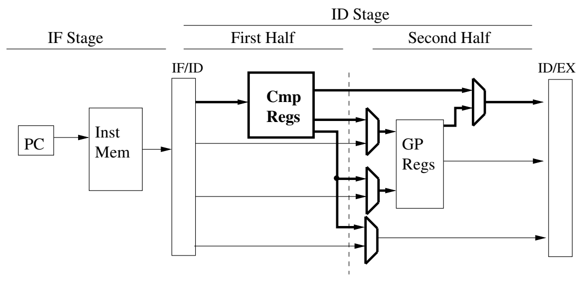


Figure 4.16: Overview of Modified Decode Stage

architecture in our experiments has a classic five-stage pipeline to which we propose adding a few minor hardware additions to support comparisons specifications. The hardware requirements for this technique are simpler than the requirements for the technique using implicit comparisons. In addition these hardware modifications are both simpler and require less storage than the hardware needed for most modern branch predictors. A new comparison register file is used to store information that indicates the values to be compared. Read and write ports for this new register file are also required, along with hardware allowing forwarding between a comparison specification instruction and a branch instruction. A separate adder, which is common in many machines, is needed for calculating the branch target address since it is represented as a displacement from the program counter. With a separate adder, the outcome of the comparison and the branch target address can be calculated in parallel.

Figure 4.16 provides a high level view of the data path access to the comparison register file. When a branch is encountered, the comparison register file is accessed in the first half of the *decode* stage. The values from the comparison register file are register numbers used to indicate which general-purpose registers hold the actual values to be used in the comparison. The general-purpose register file is accessed in the second half of the *decode* stage to get the comparison values. The comparison register file can also hold constants, provided they are small enough to fit within the size allocated for the comparison register. The values to be compared (whether the contents of a general-purpose register or a constant value) are passed to the *execute* stage. While the register numbers of the general-purpose register to be compared (or a constant value) are stored in the comparison register, the type of comparison is encoded within the branch instruction itself. Section 4.2.4 discusses the encoding of the new instructions we added to the ARM ISA in greater detail.

4.2.2 Exploiting Comparison Specifications

In this section we provide a description of the cmpspec and cbranch instructions. We also present several code examples generated with cmpspec and cbranch instructions. For these experiments, the cmpspec and cbranch instructions are generated only for those branches involving general-purpose integer registers. Conditional transfers of control involving floating-point registers are generated using conventional ARM instructions. The instructions in all the figures shown in the section, like previous sections are depicted as RTLs.

4.2.2.1 Basic Comparison Specification

Figure 4.17(a) shows a section of code generated with a typical comparison instruction and branch instruction. The instruction in line 2 is setting a condition code register (IC) by comparing the values of the contents of registers `r[2]` and `r[3]`. The branch instruction in line 3 sets the program counter to the address of the next instruction or the address associated with label `L6`, depending on the contents of the condition code register.

Figure 4.17(b) shows the code generated using cmpspec and cbranch instructions. The *c* register represents one of the new comparison registers. The instruction in line 2 stores two register numbers indicating which registers hold the values involved in the comparison. The cbranch instruction on line 3 accesses register `c[0]` to determine which values it should compare. Normal branch instructions on the ARM contain 24 bit offset fields. However, to be able to encode all the necessary information within

```
1 r[2]=MEM;
2 IC=r[2]?r[3];
3 PC=IC<0,L6;
```

(a) Original RTLs

```
1 r[2]=MEM;
2 c[0]=2,3;
3 PC=c[0]<,L6;
```

(b) New RTLs

Figure 4.17: Comparison Specification and Cbranch RTLs

Cycles

	inst	0	1	2	3	4	5	6	7
1)	load	IF	ID	EX	MEM	WB			
2)	cmp		IF	ID	stall	EX	MEM	WB	
3)	branch			IF	stall	ID	EX	MEM	WB

Figure 4.18: Pipeline Diagram for Load, Traditional Comparison and Branch

our new *cbranch* instruction, we reduced the offset field to 20 bits, which still has a range of over two million instructions and is more than sufficient to encode the branch target offset for any embedded application.

4.2.2.2 Pipelining Cmpspec and Cbranch Instructions

Figure 4.18 depicts a pipeline diagram, assuming a classical five-stage in-order pipeline, for code containing a load, comparison and branch. A stall occurs at line 2, waiting for the value generated by the load in line 1. Since the comparison instruction stalls, the branch instruction is stalled as well. The value needed by the EX stage of the comparison instruction is forwarded from the MEM stage of the load instruction after the stall.

The pipeline diagram for the code generated with cmpspec and cbranch instructions is illustrated in Figure 4.19. The cmpspec in line 2 does not stall because it only refers to the register numbers of the registers involved in the comparison and has no dependencies with the instructions that actually set the registers. By the time the cbranch completes the ID stage, the loaded value is available from the MEM stage of the load instruction to be forwarded to the EX stage of the cbranch instruction where the comparison will occur. Similarly there is forwarding from the ID stage of

	Cycles							
	inst	0	1	2	3	4	5	6
1)	load	IF	ID	EX	MEM	WB		
2)	cmpspec		IF	ID	EX	MEM	WB	
3)	cbranch			IF	ID	EX	MEM	WB

Figure 4.19: Pipeline Diagram for Load, Cmpspec and Cbranch

the cmpspec instruction to the ID stage of the cbranch instruction to indicate which registers are being compared.

A disadvantage with this new technique, like implicit comparisons, is that the branch misprediction penalty for the new cbranch instruction is greater than the misprediction penalty for a regular branch instruction. The comparison in Figure 4.18 is resolved at the end of the EX stage of the comparison instruction on line 2. If a misprediction is made, then there is a one cycle misprediction penalty. However, in Figure 4.19 the comparison is resolved at the end of the EX stage of the cbranch instruction on line 3. So there is a two cycle misprediction penalty for this pipeline when cbranch instructions are used. Our experimental results will show that on average the benefits gained from using comparison specifications outweigh the higher penalty that occurs on mispredicted cbranches.

4.2.2.3 Exploiting Loop-Invariant Code Motion

Other compiler optimizations, such as loop-invariant code motion and common subexpression elimination can be successfully performed on the new cmpspec instructions in cases where no benefit was possible using traditional comparison instructions. Cmpspecs that exist within loops can typically be moved into loop preheaders so the cost of the instruction is only incurred when the loop is entered, not on every iteration of the loop. Even though the values within the registers involved in the comparison may change from one execution of a comparison to the next, the cmpspec itself does not change. This holds true whether the comparison involves two registers or a register and a constant. Figure 4.20(a) shows a section of code with traditional

```
1 L3:
2     r[2]=MEM;
3     IC=r[1]?r[2];
4     PC=IC<0,L3;
```

(a) Original Code

```
1 L3:
2     r[2]=MEM;
3     c[0]=1,2;
4     PC=c[0]<,L3;
```

(b) Code with Cmpspec

```
1     c[0]=1,2;
2 L3:
3     r[2]=MEM;
4     PC=c[0]<,L3;
```

(c) Cmpspec Moved out of Loop

Figure 4.20: Moving Cmpspecs Out of Loops

		Cycles						
	inst	0	1	2	3	4	5	6
1)	load	IF	ID	EX	MEM	WB		
2)	cbranch		IF	ID	stall	EX	MEM	WB

Figure 4.21: Pipeline Diagram for Loop Invariant Comparison Specification

comparison and branch instructions, while Figure 4.20(b) shows code generated with a cmpspec and cbranch instruction. Figure 4.20(c) depicts the code after the cmpspec has been moved into the loop preheader. The comparison in line 3 of Figure 4.20(a) cannot be moved into the preheader since the value of `r[2]` depends on the load in line 2.

Even when the cmpspec is moved out of the loop, there is still a benefit when the cbranch stalls. Figure 4.21 shows the pipeline diagram for the code in Figure 4.20(c). Even though the cbranch has to stall waiting for the value of the comparison to become ready, it still takes one less cycle than the pipeline diagram shown in Figure 4.18 because we are only stalling for the cbranch instruction, not both a comparison instruction and a branch instruction. Getting to the branch instruction sooner, means that the branch prediction is also made sooner. Unlike the implicit comparison approach that was applied only within loops, generating cmpspec and cbranch instructions in non-loop code provides significant benefit. This benefit comes mainly from the fact that the cmpspec performs useful work in those cases where a stall would occur if an explicit comparison was generated.

```
1 L2:
2     c[0]=2,3;
3     PC=c[0]==,L6;
4     ...
5     c[0]=5,12
6     PC=c[0]!=,L5;
7     ...
8     // branch L2;
```

(a) Before Renaming

```
1 L2:
2     c[0]=2,3;
3     PC=c[0]==,L6;
4     ...
5     c[1]=5,12;
6     PC=c[1]!=,L5;
7     ...
8     // branch L2;
```

(b) After Renaming

```
1     c[0]=2,3;
2     c[1]=5,12;
3 L2:
4     PC=c[0]==,L6;
5     ...
6     PC=c[1]!=,L5;
7     ...
8     // branch L2;
```

(c) After Code Motion

Figure 4.22: Renaming Comparison Specifications

Cmpspecs are initially generated by the compiler using comparison register `c[0]` since all the cmpspecs come immediately before the cbranch that references them. When moving cmpspecs into loop preheaders, the comparison register and corresponding cbranch instructions are sometimes renamed to reference a different comparison register. If there are no remaining free comparison registers, then the cmpspec cannot be moved into the loop preheader. Figure 4.22(a) shows a section of code where the cmpspecs on lines 2 and 5 both reference comparison register `c[0]` to define the comparisons that will take place when the instructions on lines 3 and 6 are executed. While both of these cmpspecs are loop-invariant and can be moved out of the loop, one of the comparison registers needs to be renamed to avoid a conflict, which is depicted in Figure 4.22(b). Figure 4.22(c) shows the code after loop-invariant code motion has moved both cmpspecs into the preheader.

4.2.2.4 Exploiting Common Subexpression Elimination

CSE cannot typically eliminate traditional comparison instructions since they perform the actual comparison needed by the branch to determine the flow of control to follow. Figure 4.23(a) shows a section of code containing two identical comparison instructions. CSE cannot be applied to remove the second comparison in line 4 because the values in `r[2]` or `r[3]` may have changed and the comparison is not redundant. In contrast, cmpspecs are more likely to be eliminated by CSE.

Figure 4.23(b) shows the section of code generated with cmpspecs. In this case the second cmpspec in line 4 is redundant, since it only references register numbers rather than register values. Thus, it can be eliminated by CSE as shown in Figure 4.23(c).

```
1     IC=r[2]?r[3];
2     PC=IC<0,L5;
3     ...
4     IC=r[2]?r[3];
5     PC=IC>0,L5;
```

(a) Original Instructions

```
1     c[0]=2,3;
2     PC=c[0]<,L5;
3     ...
4     c[0]=2,3;
5     PC=c[0]>,L5;
```

(b) New Instructions

```
1     c[0]=2,3;
2     PC=c[0]<,L5;
3     ...
4     PC=c[0]>,L5;
```

(c) After CSE

Figure 4.23: Eliminating Redundant Comparison Specifications

In the simplest case, when there are two or more identical cmpspecs, then all but one may be deleted. However, there may be differences between cmpspecs. Modifications may sometimes be made to one or more of the cmpspec and/or cbranch instructions so that CSE may be successfully applied. It may be the case that there are two or more cmpspecs that compare the same registers, but differ only by the order of the registers. If the conditions associated with the two cmpspecs are either '=' or '$\neq$' tests, then one cmpspec can be removed without modification to the corresponding cbranch instructions. Otherwise it may be possible to modify the cbranch instruction so that one cmpspec is redundant and can be removed. In Figure 4.24(a), the cmpspecs on lines 1 and 2 (which have already been moved to the preheader of the loop), differ in the order the registers are specified. To make the cmpspec in line 2 redundant we reverse the order of the registers. To preserve the semantics of the cbranch in line 6 that accesses comparison register `c[3]`, the comparison condition must be altered from a '>' test to a '<' test as shown in Figure 4.24(b). The right hand side (RHS) of the cmpspecs in lines 1 and 2 are now identical, so CSE renames the comparison register accessed by the cbranch in line 6 from `c[3]` to `c[2]` and the cmpspec in line 2 is then removed, as shown in Figure 4.24(c).

```
1     c[2]=2,3;
2     c[3]=3,2;
3 L2:
4     PC=c[2]>,L6;
5     ...
6     PC=c[3]>,L5;
7     ...
8     // branch L2;
```

(a) Original Code

```
1     c[2]=2,3;
2     c[3]=2,3;
3 L2:
4     PC=c[2]>,L6;
5     ...
6     PC=c[3]<,L5;
7     ...
8     // branch L2;
```

(b) After Reversing Condition

```
1     c[2]=2,3;
2 L2:
3     PC=c[2]>,L6;
4     ...
5     PC=c[2]<,L5;
6     ...
7     // branch L2;
```

(c) After CSE

Figure 4.24: Reversing Branch Condition and Performing CSE

```
1     c[2]=2,0;
2     c[3]=2,1;
3 L2:
4     PC=c[2]#>,L6;
5     ...
6     PC=c[3]#<,L5;
7     ...
8     // branch L2;
```

(a) Original Code

```
1     c[2]=2,0;
2     c[3]=2,0;
3 L2:
4     PC=c[2]#>,L6;
5     ...
6     PC=c[3]#<=,L5
          ;
7     ...
8     // branch L2;
```

(b) After Modification

```
1     c[2]=2,0;
2 L2:
3     PC=c[2]#>,L6;
4     ...
5     PC=c[2]#<=,L5
          ;
6     ...
7     // branch L2;
```

(c) After CSE

Figure 4.25: Making Two Compares Use the Same Constant

Sometimes a cmpspec can be made redundant when it references the same register as another cmpspec and the constants differ by one. Figure 4.25(a) illustrates such an example. The '#' shown in the cbranch instructions means that the second value from the cmpspec should be interpreted as a constant value, not a register number. The RHS of the cmpspecs in lines 1 and 2 both reference the same register and the constants differ only by one. Since the branch in line 6, which corresponds to the cmpspec in line 2, performs a '<' test, the cbranch can be modified to a '$\leq$' test so that the value can be compared to 0 instead of 1, as shown in Figure 4.25(b). Once the branch is modified, the two cmpspecs are identical and the cmpspec in line 2 becomes redundant and will be removed, as shown in Figure 4.25(c).

```
1     c[4]=2,1;
2     PC=c[4]<=,L6;
3     ...
4     c[4]=2,1;
5     PC=c[4]#==,L5;
6     ...
```

(a) Identical Bit Pattern

```
1     c[4]=2,1;
2     PC=c[4]<=,L6;
3     ...
4     PC=c[4]#==,L5;
5     ...
```

(b) After CSE

Figure 4.26: CSE with a Register and Constant

Figure 4.26(a) shows a section of code containing two similar cmpspecs. The specification in line 1 compares the values contained in two general-purpose registers (`r[2]` and `r[1]`), while the specification in line 4 compares a register to a constant value (`r[2]` and 1). The bit pattern for both of these cmpspecs is identical, since it is the type of cbranch instruction that indicates how the bits are interpreted. The value of constants between 0-15 referenced in cmpspec instructions have the same encoding as registers 0-15. Thus, CSE can remove the cmpspec in line 4, as shown in Figure 4.26(b). This particular optimization has to be applied late in the compilation process to make sure that the comparison register referenced in lines 2 and 4 is not renamed. If comparison register `c[4]` is renamed in line 2 but not line 4 (or vice-versa), then the optimization cannot be applied.

4.2.3 Overhead of Using Comparison Specifications

For the sake of simplicity, we decided to use a scratch/nonscratch calling convention for comparison registers that is the same as the one for general-purpose registers on the ARM. When code is generated, non-scratch registers that are used within a function are saved and restored using the stack. The ARM has pseudo-instructions that store multiple general-purpose registers onto the stack, so we created a new instruction that follows the same format to load and store the new comparison registers. Comparison registers only require 16 bits to store the information needed by a cbranch instruction, so the new load or store instruction actually loads or stores two comparison registers at once. Comparison registers, like general-purpose registers, need to be saved and restored when a context switch occurs.

4.2.4 Experimental Environment

For these experiments we needed a way to simulate the execution of a program containing cmpspecs and cbranch instructions on a machine with the proposed hardware. To simulate the execution of the ARM with the hardware additions, we chose the ARM port of the SimpleScalar simulator [32]. For ARM simulations we used the default *xscale* configuration, which defines a five-stage in-order pipeline with a 128-entry bimodal branch predictor. We modified the simulator, as well as the corresponding tools (such as the GNU assembler), to incorporate our new instructions. To generate ARM code with our new instructions, we used the Very Portable Optimizer (VPO) compiler [24].

Additions had to be made to the base instruction set for the ARM to generate code containing comparison specifications. Instructions are needed to assign values for the comparison registers as well as save and restore the new comparison registers in memory. New cbranch instructions are also needed. Along with the new instructions, an encoding had to be developed for the comparison registers themselves.

Comparison registers hold register numbers for general-purpose registers containing the values to be used in the comparison. In cases where the second value to be compared is a constant and the constant is small enough, it is encoded directly in the comparison register. Figure 4.27 illustrates our encoding for the comparison registers. The bits in positions 15-12 indicate the register number for the first general-purpose register involved in the comparison. The remaining 12 bits (11-0) can either be interpreted as a constant or a register number for the second general-purpose register involved in the comparison. The ARM uses 12-bit intermediate fields in many instructions, including the original *cmp* instruction. How these bits are interpreted is encoded into the branch instruction that accesses the comparison register. For

15-12	11-5	4-0
reg num	unused	reg num
reg num	constant	

Figure 4.27: Instruction Encoding

example, the bit pattern 0001000000001000 can be used to indicate that we are comparing the values in registers `r[1]` and `r[8]` or that we are comparing the value in register `r[1]` with the constant 8. Using this encoding scheme allows more opportunities for common subexpression elimination to remove redundant cmpspecs, as shown in Figure 4.26.

A total of 4 new instructions were added to the ISA for the ARM to be able to properly use comparison specifications. These instructions are shown in Table 4.5. The cmpspec instruction assigns two values to a comparison register. The first value is a register number for the general-purpose register that contains the first value to be used in the comparison. The second value can be interpreted as either a register number for a general-purpose register that contains the second value of the comparison, or the value of a constant to be used in the comparison. The *cbr* is a cbranch instruction that references a comparison register to look up two values in general-purpose registers to be compared. The *cbri* is the same as the *cbr* instruction, except it interprets the comparison register as a register number and a constant. The *[l/s]cfd* is a CISC instruction that takes a list of comparison registers and either stores or loads them to the location in memory pointed to by the *reg* argument, which is usually the stack pointer. It is similar to other CISC ARM instructions that load or store a list of general-purpose registers. The first three instructions replace the normal ARM comparison and branch instructions.

Table 4.6 shows the bit encodings for the 4 new instructions added to the ARM ISA. For all the instructions on the ARM, bits 31-28 are used to specify a condition, or predicate determining if the instruction should execute. These bits determine the type of comparison that will be performed in branch instructions. The bits

Table 4.5: New Instructions for Comparison Specifications

New Instructions		
1	cmpspec <creg>,index1,val;	Assigns an values to be compared
2	cbr <creg><rel_op>, <label>;	Comp. register contains indices
3	cbri <creg><rel_op>, <label>;	Comp. register contains an index and a constant
4	[l/s]cfd <reg>,{register list};	CISC inst - stores/loads comp. regs to/from stack

Table 4.6: Encoding for the New ARM Instructions

New Instructions								
name	31-28	27-24	23-20	19-16	15-12	11-8	7-4	3-0
cmpspec	cond	**0110**	rrrr	vvvv	vvvv	vvvv	00**01**	cccc
cbr	cond	**101**0	aaaa	aaaa	aaaa	aaaa	aaaa	cccc
cbri	cond	**101**1	aaaa	aaaa	aaaa	aaaa	aaaa	cccc
[l/s]cfd	cond	**0110**	mmmm	mmmm	mmmm	mmmm	0L**11**	rrrr

shown in bold are the bits that represent the opcodes for the instruction. Bits 27-24 and 5-4 represent the opcode for the *cmpspec* and the *[l/s]cfd* , while bits 27-25 represent the opcode for the branch instructions *cbr* and *cbri.* Bits 19-16 (r) in the cmpspec instruction represent the register number of the first value involved in the comparison. Bits 15-8 (v) represent the second value in the comparison, which can be either a constant or another register number. Bits 3-0 (c) represent the comparison register being set. For the cbr and cbri instructions, bits 23-4 (a) are the offset used to calculate the branch target address, while bits 3-0 (c) represent the comparison register that contains the values involved in the comparison. Bit 24 (0 or 1) indicates whether the second value in the comparison register should be interpreted as a register number or a constant. For the [l/s]cfd instruction, bits 19-4 (m) indicate a mask for the 16 comparison registers that may be loaded or stored into memory, 1 bit representing each register. Bits 3-0 (r) represent the register containing the base address where the registers should be loaded or stored. Bit 6 (L) indicates whether to perform a load or store operation.

4.2.5 Encoding Problem

One problem we encountered with the developing the encoding of our instructions for this technique was that our cmpspec instructions sometimes interfered with compiler analysis that was needed for other optimization phases. A cmpspec specifies the numbers of general-purpose registers that will be used in a comparison. We first represented cmpspecs as shown in Line 1 of Figure 4.28. It was quickly discovered that this representation interfered with live variable analysis. Although we are specifying which registers are involved in the comparison, they do not have to be live at the time the cmpspec is executed. Instead, they must be live when the cbranch is executed. Remember that cmpspecs are many times moved into loop preheaders, and thus interfered with calculating correct live ranges for general-purpose registers.

```
1    c[2]=r[5],r[6];
2    c[2]=5,6;
```

Figure 4.28: Different Representations of a Cmpspec RTL

To solve this problem we modified the cmpspec RTLs so that only the index of the registers involved in the comparison are specified as shown in line 2 of Figure 4.28. This representation solved the majority of problems that arose with live variable analysis. However, it presented a new challenge. If either of the general-purpose registers indicated in the cmpspec, in this example `r[5]` or `r[6]`, are renamed after the cmpspec is generated, the references to those registers would not be renamed in the cmpspec itself since the existing analysis does not recognize them as registers. To solve this problem, we modified the cmpspec RTLs before any compiler phase that might rename registers, so that the references contained within the cmpspec would also be renamed. Once renaming is finished, the cmpspec RTLs were modified back so the references to the general-purpose registers would not affect live variable analysis for the general-purpose registers.

4.2.6 Results

Our technique using comparison specifications was tested for the same set of benchmarks from the MiBench suite of benchmarks, described in Table 4.2, that were used to test *implicit comparisons*. The benchmarks were compiled with the VPO compiler and SimpleScalar was used to simulate execution of programs on a machine that contains hardware to support comparison registers.

Code was generated using comparison specification instructions and dynamic instruction counts and cycles times were obtained. The top bars in Figure 4.29 show both the percentage of instructions saved using the new comparison specifications as well as any overhead involved with saving and restoring the comparison registers. For these experiments instruction counts were measured in micro-ops, which is the basic unit of execution on the ARM. Most ARM instructions take 1 micro-op to complete. However, some of the ARM CISC instructions, which include the newly added [l/s]cfd instruction, can take multiple micro-ops. The average savings for dynamic instructions was roughly 5.6%, while the overhead involved to save and restore the comparison register was roughly 0.9%. The greatest savings came from *adpcm* with a savings of roughly 18%. The *ispell* benchmark actually executed around 4% more instructions. One cause of this is due to the overhead involved with saving and restoring the comparison registers. The overhead for ispell was the largest of any of our tested benchmarks at around 9%. The majority of savings in dynamic instruction counts comes from loop-invariant code motion, which is about 5.3%, while the remaining savings, about 0.3%, comes from common subexpression elimination.

The bottom bars in Figure 4.29 show the percentage of execution cycles saved. Execution cycles and dynamic instruction counts do not have a one-to-one correspondence. Different instructions require a different number of cycles to complete. The cycles needed for an instruction to complete can also vary since an instruction can

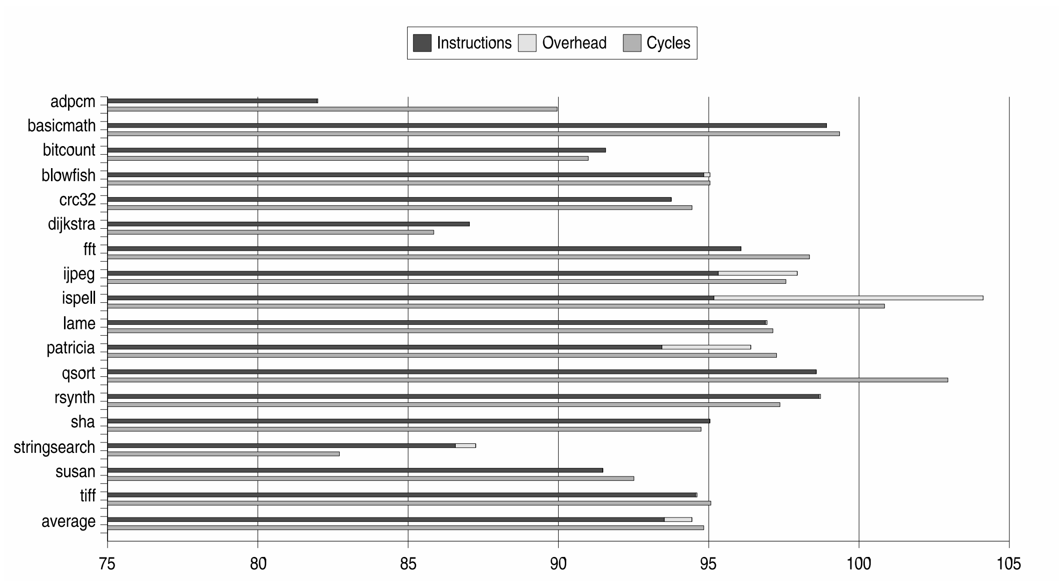


Figure 4.29: Percentage of Instructions and Cycles Executed

sometimes stall waiting for its operands to become ready. Our experiments show that a large portion of the savings in execution cycles comes from not having to stall the pipeline when a cmpspec instruction is reached, in those cases where a stall occurs using a traditional comparison instructions. A smaller percentage of the savings comes from applying optimizations, such as loop-invariant code motion and common subexpression elimination on the new cmpspec instructions, even though applying these optimizations substantially reduced the number of instructions executed. Most of the benchmarks showed a reduction in execution cycles needed for a successful program run. However, *ispell* and *qsort* both required more cycles to complete execution. The average savings for cycles executed is roughly 5.2%, ranging from a 3% loss (*qsort*) to a 17% gain (*stringsearch*). Separate tests were run to determine how much loop-invariant code motion and common subexpression elimination contributed to the improvement. The results show that about 0.20% of the improvement comes from loop-invariant code motion and 0.40% comes from common subexpression elimination. Even though loop-invariant code motion and common subexpression

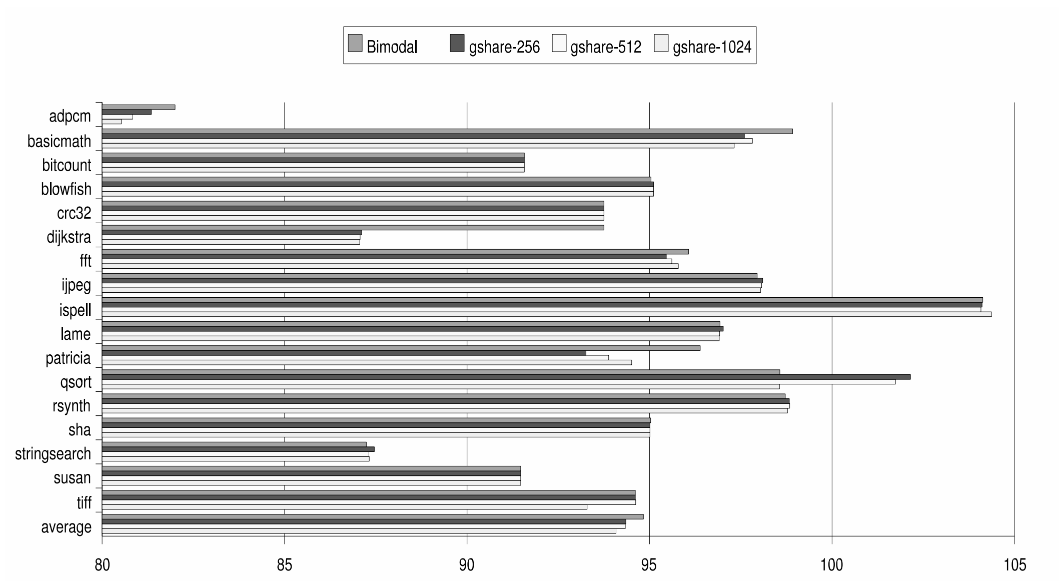


Figure 4.30: Percentage of Micro-ops Saved Using Different Prediction Methods

elimination do not have a great impact on execution cycles, fewer instructions are fetched, decoded and executed, which should reduce energy consumption.

While most of the benchmarks tested showed an improvement in both dynamic instruction counts and execution cycles, two of the benchmarks, *ispell* and *qsort*, actually required more execution cycles to complete. Analysis of the benchmarks show the main reason for this loss was the higher misprediction penalty required for the cbranch instructions. Figure 4.30 shows the percentage of dynamic instructions saved using comparison specifications when different branch predictors were used. The dynamic number of instructions (or micro-ops) and execution cycles are highly correlated, as stalls and branch prediction affect both measurements. Figure 4.31 shows the execution cycles required for each benchmark to successfully complete using the different prediction methods. We ran the experiments using the standard bimodal predictor specified in the *xscale* configuration file. We then re-ran the experiments using a gshare predictor, where we increased the second level entry table size, which in turn led to more accurate predictions overall.

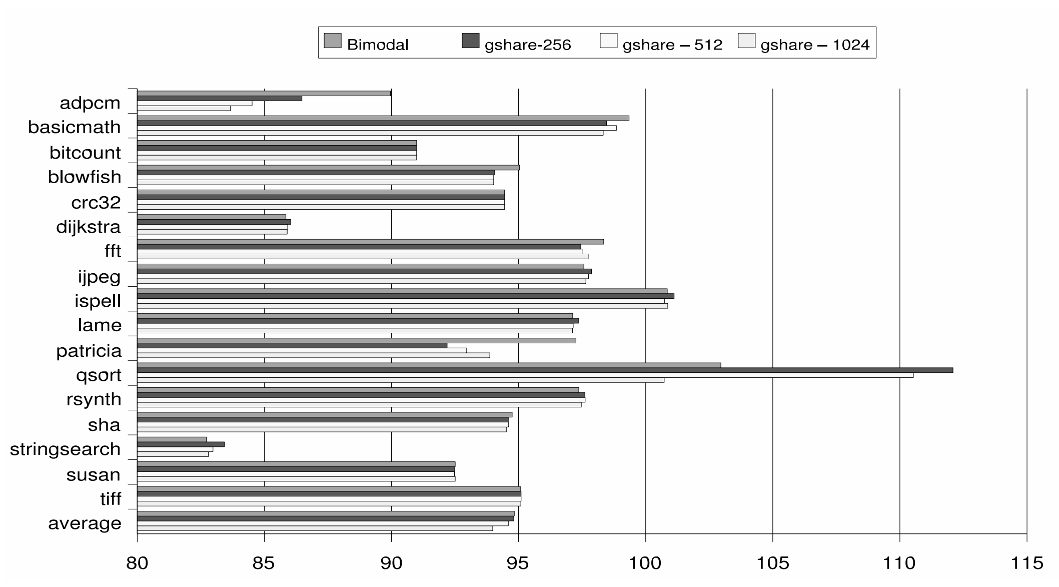


Figure 4.31: Percentage of Cycles Saved Using Different Prediction Methods

Table 4.7: Benefits Using Alternate Branch Prediction Method

	bimodal-128	gshare-256	gshare-512	gshare-1024
Micro-ops Reduced	5.6%	5.7%	5.7%	5.8%
Cycles Reduced	5.2%	5.2%	5.4%	6.0%
Misprediction Rate	10%	9.9%	8.1%	6.9%

Table 4.7 summarizes that data from Figures 4.30 and 4.31 as well as presents the misprediction rates for the benchmarks when using the different branch predictors. This table shows that using modern, more efficient branch predictors can improve the benefits gained from this technique. However, the advantages of using comparison specifications still outweighed the disadvantages of the higher misprediction penalty even with the poorer performing bimodal branch predictor.

As the pipeline diagram in Figure 4.19 shows, a cmpspec can be performed between a load and a cbranch instruction without a stall. Even though the same number of cycles are required when the cmpspec has been moved out of a loop, as shown in Figure 4.21, there is no useful work being done between the load and the cbranch instruction so a stall occurs. If no stall occurs before applying loop-invariant code motion on a cmpspec and a stall occurs on the cbranch instruction afterward, then moving the cmpspec instruction to the loop preheader does not provide any additional reduction in execution cycles.

4.3 Implicit Comparisons Vs. Comparison Specifications with Cbranches

While the two techniques discussed in this chapter both decouple the values involved in the comparison from the actual comparison itself, there are some significant differences. When the idea for implicit comparisons was first developed, some of the complications that would arise were not foreseen. As a result, there are many cases in the implicit comparison technique where comparisons could not be generated in the implicit manner. Thus, opportunities for benefits were lost.

The first difficulty with generating implicit comparisons is that once a comparison specification is encountered, implicit comparison are enabled for all assignments to a specified register until the implicit comparisons are disabled (which occurs at context switches and upon entering a new function). This means that assignments of a specific register may have an implicit comparison associated with them, even though the live range of this register does not reach a branch instruction, which can lead to greater energy consumption. To minimize the number of *extra* comparison performed, implicit comparison were only generated inside loops. Another difficulty with implicit comparisons is that comparison registers have a one-to-one correspondence to general-purpose registers, which is necessary since it is the assignment to a specific register that triggers an implicit comparison. However, this makes renaming comparison registers difficult, since corresponding general-purpose registers have to be re-named as well.

With implicit comparisons, the comparison specification contains indices indicating the location of the values to be used in the comparison as well as the relational operator that determines the *type* of comparison that will occur. To determine what type of comparison will be performed we must know which of the registers involved in the comparison is set *last* before the branch is reached. If analysis cannot determine this, then it may not be possible to generate the comparison in the implicit manner. Finally when generating code with implicit comparisons, we may have multiple assignments to registers that occur within a single live range of a register that, if generated in the implicit manner, would have conflicting comparison specifications. In such cases, some of the comparisons must be generated in the explicit manner, to avoid conflicts.

While there are benefits to be gained from using implicit comparisons, the difficulties in generating implicit comparison instructions convinced us that we were missing opportunities. To resolve this, another technique was designed using the same idea of decoupling the comparison values from the actual comparison, this time designed to avoid the problems that we encountered with implicit comparisons. This technique, previously discussed in this chapter, generates code with cmpspec and cbranch instructions. This approach provides us with a greater benefit than implicit comparisons and avoids the difficulties involved with generating implicit comparisons. In general, the compiler modifications needed for cmpspecs with cbranches were much simpler that those needed for implicit comparisons. Another benefit with using the cbranch technique is that the hardware needed is simpler than the hardware needed to support implicit comparisons.

By comparing the results from both techniques we see that code generated with cmpspec and cbranch instruction performs better than code generated with implicit comparisons. However, the research involved to develop implicit comparisons was not wasted, as many of the problems encountered, helped us to develop the cbranch technique that was simultaneously simpler and more beneficial than the implicit comparison technique.

CHAPTER 5

FUTURE WORK

We believe that there is still work to be done for all three techniques that could improve the benefits presented in this dissertation. The following section discusses ideas that should lead to further improvements in reducing the cost of traditional transfers of control.

5.1 Condition Merging

For the condition merging technique one area to explore is the use of more aggressive analysis to detect when speculatively executed loads would not introduce exceptions. We perform condition merging in a compiler back-end using a representation that is equivalent to machine instructions. One advantage of performing condition merging in a back-end is that more accurate estimates of the performance benefits can be made, which is very important when applying transformations based on path profile data. However, a disadvantage of performing a transformation in a back-end is that much of the semantic information which one could use to determine if a load instruction can cause an exception is not immediately available. We conservatively merged sets of conditions, which required loads to be speculatively executed, only when these loads could not introduce new exceptions. By performing more aggressive analysis, one should be able to detect more sets of conditions to be merged. This will be particularly useful for merging conditions that cross loop boundaries.

Value range analysis may allow condition merging to be applied more frequently. Consider if a and max in Figure 3.15 were signed instead of unsigned variables. max is guaranteed to be nonnegative in this example since it is initialized to zero and it is only updated with a value that is greater than itself. Merging conditions in this case can be applied using an unsigned $\leq$ operator. If either $a[i]$ or $a[i+1]$ is negative, then the merged condition will fail and the original conditions will be tested.

We also found that the merging of one set of conditions may inhibit the merging of another set. Code is duplicated and the paths within a function are modified when conditions are merged. This code duplication invalidates the path profile data on which condition merging is based. Thus, merging a set of conditions is not currently allowed whenever the control flow changed in the sub-paths associated with the set of conditions to be merged. With careful analysis one may be able to infer new path frequency measurements for these duplicated paths.

We believe that condition merging may be useful in other settings. Condition merging may be a very good fit for run-time optimization systems, which optimize frequently executed paths during the execution of a program. Condition merging may also be useful for low power embedded systems processors where architectural support for ILP is not available.

5.2 Implicit Comparisons

When multiple comparisons that reference the same register within a single live range occur in a loop, then only one of these comparisons can be chosen to be made implicit. Likewise, sometimes there are not enough registers to move all the comparisons that were candidates to become implicit out of a loop. In these cases simple static heuristics are used to determine which comparison would be most beneficial. We believe better choices can be made by using program profiling information.

In testing implicit comparisons, we used a simple 2-bit correlating predictor, built into EASE, to perform branch prediction. More accurate predictors could be used and should increase the benefit of implicit comparisons due to fewer misprediction stalls

5.3 Comparison Specifications

In code generated with cmpspec and cbranch instructions, we believe that profiling could be used to better guide compiler optimizations, such as loop-invariant code motion, when they are applied to cmpspecs. Occasionally there are cases when saves and restores of comparison registers are executed more often than the cbranch instructions that use these registers, such as what occurred in *ispell.* Profiling would allow us to detect these cases and refrain from applying loop-invariant code motion on cmpspecs in cases where it will not help reduce the number of cycles executed.

We believe that with better analysis there are more opportunities to be gained by performing CSE on cmpspecs. For example, consider two cmpspecs (where one dominates the other) that compare two differing sets of registers as illustrated in Figure 5.1(a). The cmpspecs in lines 2 and 6 are similar but they compare different registers. However, since the live range of registers 5 and 7 do not overlap, register 7 from the second comparison can be renamed to register 5 as shown in Figure 5.1(b). Now the two cmpspecs are identical and the one in line 6 can be removed by CSE as shown in Figure 5.1(c).

We also believe that using comparison specifications would be even more beneficial with 16-bit architectures, like the Thumb, since there are fewer bits available to encode the comparison and branch instructions. Comparison specifications could be used to encode information about the branch instruction that would facilitate branching. Information about the values involved in the comparison and the type of comparison could be encoded in a comparison register. The comparison register could also be used to extend the range of instructions that a branch could reach since

```
1    r[5]=MEM;
2    c[3]=4,5;
3    PC=c[3]<=,L6;
4    //r[5] dies
5    r[7]=MEM;
6    c[3]=4,7;
7    PC=c[3]==,L5;
8    ...
```

(a) Compares Involving Different Registers

```
1    r[5]=MEM;
2    c[3]=4,5;
3    PC=c[3]<=,L6;
4    // r[5] dies
5    r[5]=MEM;
6    c[3]=4,5;
7    PC=c[3]==,L5;
8    ...
```

(b) After Renaming

```
1    r[5]=MEM;
2    c[3]=4,5;
3    PC=c[3]<=,L6;
4    // r[5] dies
5    r[5]=MEM;
6    PC=c[3]==,L5;
7    ...
```

(c) After CSE

Figure 5.1: CSE by Renaming Registers

the entire branch target displacement does not have to be encoded within the branch instruction itself.

CHAPTER 6

CONCLUSIONS

In this dissertation we describe three techniques that reduce the costs of conditional transfers of control by focusing on the comparison instruction. The first techniques performs condition merging on a conventional scalar processor. We replaced the execution of two or more branches with a single branch by merging conditions. Path profile information is gathered to determine the frequency that paths are executed in the program. Sets of conditions that can be merged are detected and the benefit of merging each set is estimated. The control flow is then restructured to merge the sets of conditions deemed beneficial. The results show that significant reductions can be achieved in the number of branches performed for non-numerical applications. We showed an average reduction in the number of branches executed, by approximately 16% and there were about 6% fewer instructions executed. Execution time for the benchmark programs was reduced by roughly 3%, and the average fetch cost decreased by almost 5%.

We have also shown that the reduction of integer scalar variables to bits within a flag variable can be automated and results in opportunities to merge conditions. We were also able to merge conditions comparing multiple variables to constants or invariant values. Unlike prior work on control CPR, we were able to accomplish our techniques through innovative use of available instructions on a conventional scalar processor. Finally, we have shown there are benefits to be obtained by merging conditions in paths that are not the most frequent. In many cases we were able to generate a *breakeven* path that allows a set of conditions to be merged when the *win* path was not the most frequently executed path upon reaching the first condition.

The second and third techniques both reduce the costs associated with traditional transfers of control by decoupling the definition of the values to be compared from the comparison itself. Unlike typical comparison instructions, comparison specifications can usually be moved outside of loops by loop-invariant code motion because they do not have dependencies with the instructions that produce the values used in the comparison. Likewise, redundant comparison specification instructions can be removed in many cases when CSE cannot be applied to typical comparison instructions. The final technique discussed in this dissertation, that pairs comparison specifications with cbranch instructions, is superior to the implicit comparison technique. While both techniques are similar, the hardware needed to perform comparison specifications with cbranches is simpler than the hardware needed for implicit comparisons. In addition, the compiler is much simpler for comparison specifications and cbranches. Finally, comparison specifications with cbranches produce better results than implicit comparisons.

Both techniques, unlike explicit comparisons, have less comparison instructions to fetch and decode, allowing the branch instruction to be accessed and predicted earlier in the instruction stream. The code generated with implicit comparisons resulted in a 5% average reduction in dynamic instructions and an average 4% savings in execution cycles. The code generated with cmpspec and cbranch instructions shows better benefits than code generated with implicit comparisons, with an average of 5.6% improvement for dynamic instruction counts and 5.2% improvement for execution cycles. Much of the improvement in execution cycles occurs because the processor does useful work during the cmpspec, while it may stall during a conventional comparison instruction.

This dissertation presents three novel approaches that reduce the costs associated with traditional transfers of control. All of these techniques focus on the comparison portion of a conditional transfer of control, which has been largely overlooked in the past. The first technique, condition merging, is able to merge conditions involving multiple variables on a conventional scalar processor. It also obtains benefits along paths that are not the most frequently executed paths.

The second two techniques were targeted for embedded systems and both reduce the costs associated with traditional transfers of control by decoupling the specification of the values to be compared with the actual comparison. Although the third technique outperforms the second technique and was simpler to implement, the effort spent on the second technique was not wasted, as the lessons learned led to the creation of the third technique. Execution cycles are reduced in many cases because the processor is able to perform useful work during the cmpspec instruction. We also showed that optimizations that cannot normally be applied to traditional comparison instructions can be applied to cmpspecs since the cmpspec instructions do not have a dependency on the instructions that produce the values to be compared.

REFERENCES

[1] Magnolis G. H. Katevenis. Reduced instruction set computer architectures for VLSI. Technical report, The MIT Press, Cambridge, Massachusetts, 1985.

[2] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, second edition, 1996.

[3] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[4] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pages 128–137. ACM SIGARCH/SIGOPS/SIGPLAN, October 1996. Published as Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News, volume 24, number Special.

[5] J. Kalamatianos and D. R. Kaeli. Predicting indirect branches via data compression. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 272–284, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.

[6] Daniel Jimenez and Calvin Lin. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems*, volume 20, pages 369–397. ACM, November 2002.

[7] Jack W. Davidson and David B. Whalley. Reducing the cost of branches by using registers. In *Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News*, pages 182–191. ACM, June 1990. Published as Proc. 17th Annual Symposium on Computer Architecture (17th ISCA'90), Computer Architecture News, volume 18, number 2.

[8] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, USA, 1971. Transformations.

[9] J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software, Practice and Experience*, 9(3):219–226, March 1979.

[10] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[11] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 56–66, La Jolla, CA, June 1995. ACM Press.

[12] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 146–158, New York, June 15–18 1997. ACM Press.

[13] Gang-Ryung Uh and David B. Whalley. Effectively exploiting indirect jumps. *Software Practice and Experience*, 29(12):1061–1101, October 1999.

[14] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM, ACM, January 1983.

[15] J. C. H. Park and M. S. Schlansker. *On predicated execution.* Hewlett Packard Laboratories, 1991.

[16] R. Towle. *Control and Data Dependence for Program Transformations.* Ph.D. thesis, Computer Science Dept., University of Ill., Urbana-Champaign, Ill., March 1976.

[17] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, 1992.

[18] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. *The superblock: an effective structure for VLIW and superscalar compilation.* Center for Reliable and High-Performance Computing, Urbana, Il, 1992.

[19] Michael Schlansker, Scott Mahlke, and Richard Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 155–168, Atlanta, Georgia, May 1–4, 1999. ACM Press.

[20] Michael Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO, IEEE Computer Society Press.

[21] J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. In *The Journal of Supercomputing*, volume 7, pages 181–228. January 1993.

[22] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, September 1990. Published in ACM SIGARCH Computer Architecture News Vol. 18, No.3.

[23] Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Pub. Co., Reading, MA, USA, 1995.

[24] Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, GA, USA, June 1988. ACM Press.

[25] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[26] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 130–141, Montreal, Canada, 17–19 June 1998. ACM Press.

[27] Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Efficient and effective branch reordering using profile data. volume 24, pages 667–697, November 2002.

[28] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[29] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[30] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[31] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman, Harlow, Essex CM20 2JE, England, second edition, 2000. Also available in Japanese translation, *ARM Processor*, C Q Publishing Co., Ltd. ISBN 4-7898-3351-8.

[32] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

BIOGRAPHICAL SKETCH

William C. Kreahling

The author was born on April 18, 1969. He received an A.A. degree in Business from Palm Beach Community College in August of 1992. He received his B.S. degree in Computer Science (Magna Cum Laude) for Appalachian State University in December of 1996. He received his Master of Science in Computer Science in August of 1999 from Appalachian State University, where his thesis dealt with profile assisted register allocation. After Graduating with his Master of Science he was employed as faculty at Appalachian State University from 1999 to 2001. His research interests include: compilers, computer architecture, and parallel computing.