Reducing the Cost of Conditional Transfers of Control by Using Comparison Specifications

William Kreahling
Western Carolina University
wkreahling@email.wcu.edu

Stephen Hines David Whalley
Gary Tyson
Florida State University
{hines,whalley,tyson}@cs.fsu.edu

Abstract

A significant portion of a program's execution cycles are typically dedicated to performing conditional transfers of control. Much of the research on reducing the costs of these operations has focused on the branch, while the comparison has been largely ignored. In this paper we investigate reducing the cost of comparisons in conditional transfers of control. We decouple the specification of the values to be compared from the actual comparison itself, which now occurs as part of the branch instruction. The specification of the register or immediate values involved in the comparison is accomplished via a new instruction called a *comparison specification*, which is loop invariant. Decoupling the specification of the comparison from the actual comparison performed before the branch reduces the number of instructions in the loop, which provides performance benefits not possible when using conventional comparison instructions. Results from applying this technique on the ARM processor show that both the number of instructions executed and execution cycles are reduced.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Optimizations; D.4.7 [*Organization and Design*]: Real-time and Embedded Systems

General Terms Algorithms, Measurement, Design, Performance, Experimentation

Keywords Branch, Compiler, Comparison, Optimization

1. Introduction

A significant portion of executed instructions are dedicated to conditional transfers of control. These instructions consume a large number of cycles, cause pipeline flushes when they are mispredicted and generally inhibit other code improving transformations. A conditional transfer of control can be broken into three distinct parts. First, the comparison determines if the branch is to be *taken* or *not-taken*. Second, the branch target address is calculated. Third, the actual transfer of control takes place. While much research has been done on reducing the cost of branch instructions and the cost of calculating the branch target address, the comparison portion has received less scrutiny. This paper presents an approach that removes comparison instructions and replaces them with comparison specifications using simple architectural modifications.

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. This enables the branch instruction to contain more bits to specify the branch displacement. The disadvantage is that two instructions have to be fetched and decoded, delaying the branch instruction fetch and resolution by at least one cycle, and more if the comparison instruction stalls waiting for source operand values. When a comparison instruction is stalled (on an in-order, embedded processor), the branch instruction is also stalled and no prediction can be made until the values required by the comparison instruction are available. In contrast, execution is not typically stalled when a branch instruction is reached, even when the values required for the comparison are unavailable. Instead a branch prediction is made and the next *predicted* instruction is fetched and executed. This eliminates the pipeline stalls as long as the prediction is correct and the conditional values are available at branch resolution, which occurs later in the pipeline.

A conditional transfer of control can also be implemented using a single instruction that performs both the comparison and branch. The advantage of this scheme 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 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 and/or comparisons to an arbitrary constant cannot be made. Most embedded processors, including the ARM processor used in this study, avoid this trade-off by relying on two instructions, a comparison followed by a branch, to handle conditional control flow operations.

We propose a technique that uses a single compare and branch instruction along with a new *comparison specification* instruction that decouples the specification of the values to be compared with the comparison itself. This new technique retains the advantages from both of the other two approaches without incurring their disadvantages. Comparison specifications define the values that are to be compared, but the comparison does not occur until the com-

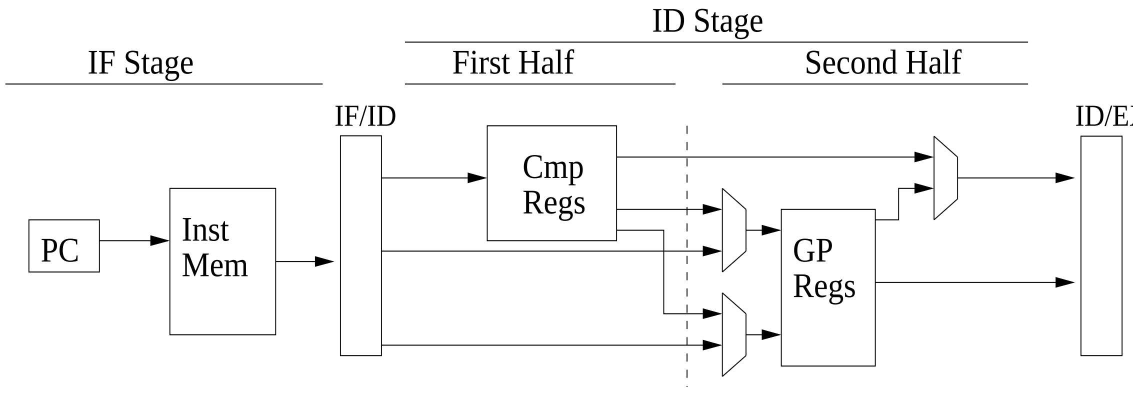

Figure 1. Overview of Modified Decode Stage

pare and branch instruction is reached. With this technique, the branch instruction is often reached sooner than approaches using separate comparison and branch instructions. Because comparison specifications 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 comparison specifications, in cases that would not be possible with traditional comparison instructions. Due to less information being encoded within our single compare and branch instructions, our technique does not significantly decrease the branch target address range or limit the values that can be compared. While this approach could be applied to a dynamically scheduled machine, it is more likely that a comparison instruction would be scheduled in parallel with other instructions, reducing the benefits that might be obtained. However, this work is ideally suited for many statically scheduled embedded systems, as we are removing many comparison instructions from the critical path. 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.

2. Related Work

There has been a lot of work investigating branch cost reduction over the years. Most of this work has focused on reducing the cost of the branch instruction itself. A very simple and straightforward technique to reduce the cost of branch stalls is the *delayed branch* [8]. In architectures that support delayed branches, one or more instructions immediately following a branch instruction are executed regardless of whether or not the branch is taken. Delayed branches have recently become less attractive than the use of branch target and prediction buffers due to processors supporting multi-issue and more complex pipelines.

Branch prediction is a technique that decides which instruction after a branch should be fetched when the branch instruction is encountered but before the outcome of the comparison is known. This decision is usually based on the past behavior of executed branches. Using branch prediction, execution of the program does not need to stall when a branch instruction is reached. If an incorrect decision is made about which instruction follows the branch, then a misprediction penalty is required to flush the incorrect instructions from the pipeline and fetch the correct instructions. Hardware branch prediction successfully reduces branch costs when it accurately predicts the direction a branch will take [8]. There are many different methods for branch prediction, ranging from single branch predictors, correlating predictors, tournament predictors, Markov predictors, to lesser known neural methods for branch prediction [8, 10, 9]. Even when hardware prediction successfully predicts the outcome of a branch, there may be delays while the branch target address is calculated. A branch target buffer acts as a small cache containing branch target addresses and when given the address of a branch, will return the actual target address [11, 8].

The use of predication can also reduce branch costs. Predication is the conditional execution of an instruction based upon a boolean source operand, called the *predicate*. Predicated instructions are fetched regardless of the value of the predicate. When the predicate evaluates to *false,* the effects of the predicated instructions are nullified [14]. A compiler optimization called *if-conversion* eliminates conditional branches by converting control dependencies into data dependencies [1]. While branch instructions are eliminated, the instruction that sets the predicate register remains.

Other techniques reduce branch costs by trying to avoid certain conditional branches entirely. Methods such as loop unrolling [1, 5] and loop unswitching [1, 12] involve loop transformations that can reduce loop overhead and may reduce the number of branches executed. Methods like intraprocedural and interprocedural conditional branch elimination duplicate code along paths where branch results can be statically determined [13, 4]. All of these approaches result in significant code growth, which may not be desirable for embedded applications.

3. New Hardware

The underlying architecture used for these experiments is the ARM processor [6]. We chose the ARM since this is a popular embedded ISA for which there exists a commonly used simulator, SimpleScalar [2]. The ARM also has separate comparison and branch instructions and thus was a good fit for our technique. The baseline ARM architecture in our experiments has a classic five-stage pipeline to which we propose adding a few minor hardware additions to support comparisons specifications. 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 1 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 obtain 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.

4. Exploiting Comparison Specifications

In this section we provide a description of the *cmpspec* (comparison specification) instruction and a new instruction we call a *cbranch*, where a comparison and branch are performed by the same instruction. 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 paper are depicted as register transfer lists (RTLs). RTLs are machine and language-independent representations of machine-specific instructions, used by many compilers as an intermediate language, including GCC [15] and VPO (Very Portable Optimizer)[3], which is the compiler used to conduct this research.

4.1 Basic Comparison Specification

Figure 2(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 sequential instruction after the branch or the address associated with label `L6`, depending on the contents of the condition code register.

Figure 2(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. In our new *cbranch* instruction, we reduced the offset field by 4 bits to make room for the reference to a c register. The offset in the *cbranch* instructions is still more than sufficient to encode the branch target address for the vast majority of embedded applications, including all of the benchmarks used in this paper.

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

(a) Original RTLs (b) New RTLs

Figure 2. Comparison Specification and Cbranch RTLs

4.2 Pipelining Cmpspec and Cbranch Instructions

Figure 3 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.

	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 3. Pipeline Diagram for Load, Traditional Comparison and Branch

The pipeline diagram for the code generated with cmpspec and cbranch instructions is illustrated in Figure 4. 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 and is forwarded to the EX stage of the cbranch instruction where the comparison will occur. Similarly there is forwarding from the ID stage of the cmpspec instruction to the ID stage of the cbranch instruction to indicate which registers are being compared.

	Cycles							
inst	0	1	2	3	4	5	6	7
1) load	IF	ID	EX	MEM	WB			
2) cmpspec		IF	ID	EX	MEM	WB		
3) cbranch			IF	ID	EX	MEM	WB	

Figure 4. Pipeline Diagram for Load, Cmpspec and Cbranch

In a conventional system with separate comparison and branch instructions, it is possible to resolve a branch prediction once the condition is determined (when the branch instruction is still in the decode stage). Some processor implementations will wait for the branch to execute, but others use additional circuitry to perform early branch resolution. However, with our modification, the condition is evaluated when the branch is in the execute stage of the pipeline, so early branch resolution is not possible. To account for this, we increase the branch misprediction penalty for the new cbranch instruction by one cycle. This accurately reflects the performance when compared to an implementation of the ARM processor that incorporates early branch resolution, while underestimating the relative performance of our enhancement when compared with implementations of the ARM processor that do not incorporate early branch resolution. resolved at the end of the EX stage of the comparison instruction on line 2. If a misprediction is made, there is a one cycle misprediction penalty. However, in Figure 4 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.

4.3 Exploiting Loop-Invariant Code Motion

Other compiler optimizations, such as loop-invariant code motion and common subexpression elimination can now 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 5(a) shows a section of code with traditional comparison and branch instructions, while Figure 5(b) shows code generated with a cmpspec and cbranch instruction. Figure 5(c) depicts the code after the cmpspec has been moved into the loop preheader. The comparison in line 3 of Figure 5(a) cannot be moved into the preheader since the value of `r[2]` depends on the load in line 2.

```
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) With Cmpspec

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

(c) Cmpspec Out of Loop

Figure 5. Moving a Cmpspec Outside of a Loop

When the cmpspec is moved out of the loop, there is still a benefit when the cbranch stalls. Figure 6 shows the pipeline diagram for the code in Figure 5(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 3 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. In addition, less energy is consumed since fewer instructions are being fetched and executed.

Cycles

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

Figure 6. Pipeline Diagram for Loop Invariant Comparison Specification

Cmpspecs are initially generated by our compiler using comparison register `c[0]` since all the cmpspecs are placed 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 7(a) shows a section of code where the cmpspecs on lines 2 and 5 both reference comparison register `c[0]` to define the comparison 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, as depicted in Figure 7(b), to avoid a conflict, to avoid a conflict. Figure 7(c) shows the code after loop invariant code motion has moved both cmpspecs into the preheader.

```
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 7. Renaming Comparison Specifications

4.4 Exploiting Common Subexpression Elimination

Common subexpression elimination (CSE) is a code improving transformation that can eliminate instructions that compute values that are already available. CSE cannot typically eliminate the traditional comparison instructions since they perform the actual comparison needed by the branch to determine the flow of control to follow. Figure 8(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 8(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 8(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 Insts.

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

(b) New Insts.

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

(c) After CSE

Figure 8. 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 9(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 9(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 9(c).

Sometimes a cmpspec can be made redundant when it references the same register as another cmpspec and the constants referenced differ by one. Figure 10(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 line 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

```
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) Reversed 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 9. Reversing a Branch Condition and Performing CSE

in Figure 10(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 10(c).

```
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 10. Making Two Cmpspecs Use the Same Constant

Figure 11(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 11(b).

```
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 11. Applying CSE to Cmpspecs with a Register and a Constant

5. 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 and 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.

6. 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 [2]. 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 [3].

To perform code improving transformations involving comparison specifications, the base instruction set for the ARM had to be enchanced. Instructions were needed to assign values for the comparison registers as well as save and restore the new comparison registers in memory. New cbranch instructions were 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 12 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 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.

15-12	11-4	3-0
reg num	unused	reg num
reg num	constant	

Figure 12. Encoding for Comparison Registers

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 1. The *movc*, which is the cmpspec instruction, assigns two values to a single 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.

New Instructions	
1	movc <creg>,index1,val;
2	cbr <creg><rel_op>, <label>;
3	cbri <creg><rel_op>, <label>;
4	[l/s]cfd <reg>,{register list};

Table 1. New Instructions to Support Comparison Specifications

7. Implementation Issues

A problem we encountered when 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 13. 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 13. 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 13. 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.

Traditional comparisons and branches were converted to code using comparison specifications late in the compilation process. in the compilation process. This was done so that comparison specifications did not interfere with any other existing code improving transformations. Once the new comparison specification transformation was performed, other code improving transformations, such as loop-invariant code motion and common subexpression elimination were re-run to determine if there were any new opportunities for optimization. The steps taken for the comparison specification code improving transformation is shown (as pseudo-code) in Figure 14.

8. Results

Our technique using comparison specifications was tested for a large subset of benchmarks from the MiBench suite of benchmarks [7], described in Table 2. At present, we are not able to use the remaining benchmarks from the MiBench Suite under the SimpleScalar ARM port due to configuration problems with the benchmarks. 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.

```
1  Transform conditional transfers of control
       to use comparison specifications
2  Perform live variable analysis
3  Perform loop invariant code motion
4  Modify Cmpspec and Cbranches to enable CSE
5  While opportunities for CSE exist do
6     perform CSE
7  Translate RTLs to assembly code
```

Figure 14. Pseudo-code for Comparison Specification Code Improving Transformation

Name	Description
adpcm	adaptive pulse 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

Table 2. Benchmarks Tested

Code was generated using comparison specification instructions and dynamic instruction counts and cycle times were obtained. The top bars in Figure 15 show both the percentage of instructions saved using the new comparison specifications as well as the the 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 both the existing [ld/st]m instruction and 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 increase 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 15 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 sometimes stall waiting for its operands to become ready. Our experiments show that a large portion of the savings in execution

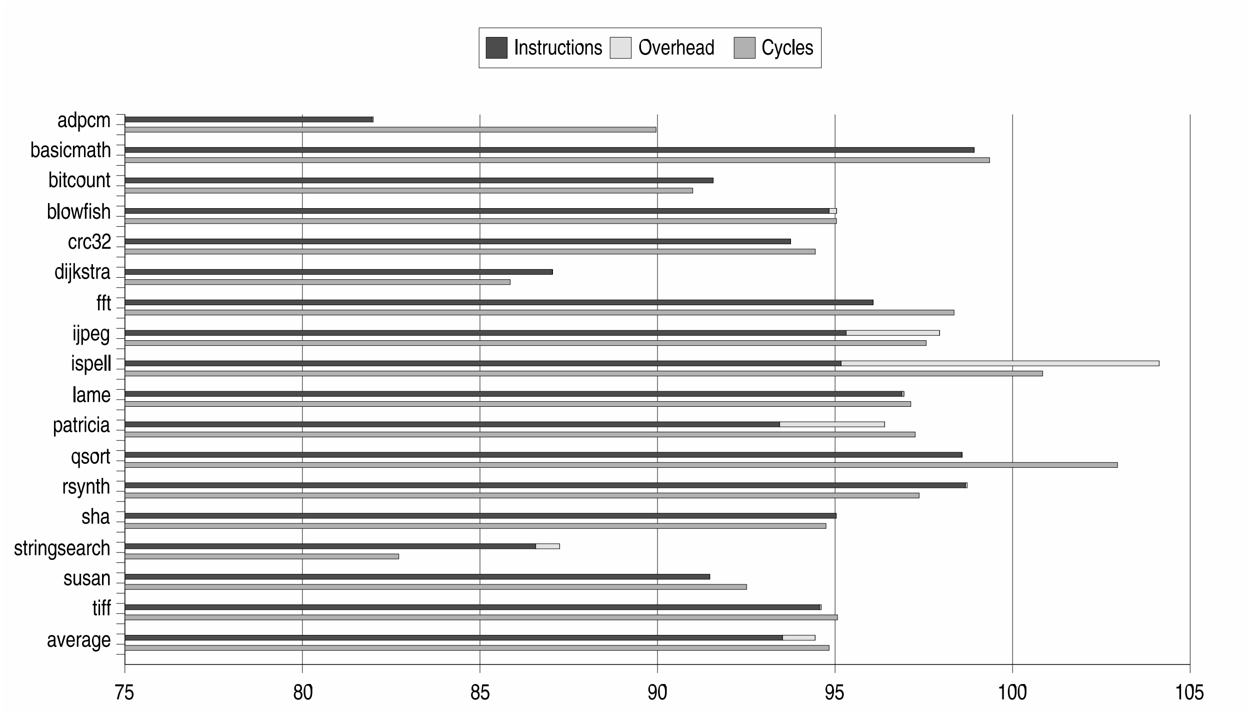

Figure 15. Percentage of Instructions and Cycles Executed

cycles comes from not having to stall the pipeline when a cmpspec instruction is reached, in those cases where a stall does occur 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 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. Table 3 compares the average savings in instructions, cycles and the misprediction rates for the benchmarks using a bimodal predictor and a gshare predictor. For the gshare predictor we used second level table sizes of 256, 512 and 1024 entries with 7 bits of global history. 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 since the number of cycles was significantly reduced. The reason for the overall performance benefit is that branch misprediction rates are relatively low and there is often a cycle saved using comparison specifications each time a conditional transfer of control is encountered.

	bimodal	gshare		
	128	256	512	1024
Instructions 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 3. Benefits Using Alternate Branch Prediction Method

9. Future Work

The pipeline design studied in this paper computes the condition when the branch instruction executes. This does not allow early resolution of the branch instruction. However, the condition test can occur as soon as the operands are available enabling the test to overlap with instructions that modify registers specified in the conditional test. By computing tests any time registers involved in the condition specification are modified, it may be possible to implement early branch resolution, thereby reducing the branch misprediction penalty.

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 [causing the use of nonscratch comparison registers] 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 16(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 16(b). Now the two cmpspecs are identical and the one in line 6 can be removed by CSE as shown in Figure 16(c).

```
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 With 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 16. CSE by Renaming Registers

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

10. Conclusions

While most techniques used to reduce the cost of conditional transfers of control focus on the branch, this paper presents a novel approach that reduces the cost by focusing on the comparison. The specification of the comparison is decoupled from the actual comparison of the values. In many cases, execution cycles are decreased since the processor does useful work during the cmpspec, while it may stall during a conventional comparison instruction. In addition, conventional compiler optimizations can be more easily applied on cmpspecs than conventional comparisons. Unlike comparison instructions, cmpspecs 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 cmpspec instructions can be removed in many cases when CSE cannot be applied to typical comparison instructions. The results show significant reduction in execution cycles of 5.2% and a 5.6% reduction in dynamic instructions.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This research was supported in part by NSF grants EIA-0072043, CCR-0208892, CCR-0312493, and CCF-0444207.

References

[1] ALLEN, F. E., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, R. Rustin, Ed. Prentice-Hall, Englewood Cliffs, NJ, USA, 1971, pp. 1–30.

[2] AUSTIN, T., LARSON, E., AND ERNST, D. SimpleScalar: An infrastructure for computer system modeling. *Computer 35*, 2 (Feb. 2002), 59–67.

[3] BENITEZ, M. E., AND DAVIDSON, J. W. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, USA, June 1988), ACM Press, pp. 329–338.

[4] BODÍK, R., GUPTA, R., AND SOFFA, M. L. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation* (New York, June 15–18 1997), vol. 32, 5 of *ACM SIGPLAN Notices*, ACM Press, pp. 146–158.

[5] DONGARRA, J. J., AND HINDS, A. R. Unrolling loops in FORTRAN. *Software, Practice and Experience 9*, 3 (Mar. 1979), 219–226.

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

[7] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., AND BROWN, R. B. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization* (December 2001).

[8] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach.*, second ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1996.

[9] JIMENEZ, D., AND LIN, C. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems* (Nov. 2002), vol. 20, ACM, pp. 369–397.

[10] MCFARLING, S. Combining branch predictors. Tech. Rep. TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[11] MCFARLING, S., AND HENNESSY, J. Reducing the cost of branches. In *Proc. 13th Annual International Symposium on Computer Architecture, Computer Architecture News* (June 1986), ACM, pp. 396–403. Published as Proc. 13th Annual International Symposium on Computer Architecture, Computer Architecture News, volume 14, number 2.

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

[13] MUELLER, F., AND WHALLEY, D. B. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* (La Jolla, CA, June 1995), ACM Press, pp. 56–66.

[14] PARK, J. C. H., AND SCHLANSKER, M. S. *On predicated execution*. Hewlett Packard Laboratories, 1991.

[15] STALLMAN, R. M. Using and porting the GNU compiler collection, Feb. 22 2001.