3.6.5 Implementing Conditional Branches with Conditional Control

The most general way to translate conditional expressions and statements from C into machine code is to use combinations of combinations of conditional and unconditional jumps. (As an alternative, we will see in Section 3.6.6 that some conditionals can be implemented by conditional transfers of data rather than control.) For example, Figure 3.16(a) shows the C code for a function that computes the absolute value of the difference of two numbers. The Function also has a side effect of incrementing one of two counters, encoded as global variables lt\_cnt and ge\_cnt. Gcc generates the assembly code shown as Figure 3.16(c). Our rendition of the machine code into C is shown as the function gotodiff\_se (Figure 3.16(b)). It uses the goto statement in C, which is similar to the unconditional jump of assembly code. Using goto statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of machine code. We call this style of programming “goto code”

In the goto code (Figure 3.16(b)), the statement goto x\_ge\_y on line 5 causes a jump to the label x\_ge\_y(since it occurs when x >= y) on line9. Continuing the execution from this point, it completes the computations specified by the else portion of function absdiff\_se and returns. On the other hand, if the test x >= y fails, the program procedure will carry out the steps specified by the if portion of absdiff\_se and return.

The assembly-code implementation first compares the two operands, setting the condition codes. If the comparison result indicates that x is greater than or equal to y, it then jumps to a block of code starting at line 8 that increments global variable ge\_cnt, computes x-y as the return value, and returns. Otherwise, it continues with the execution of code beginning at line 4 that increments global variable lt\_cnt, computes y-x as the return value, and returns. We can see, then, that the control flow of the assembly code generated for absdiff\_se closely follows the goto code of gotodiff\_se.

The general form of an if-else statement in C is given by the template

If (test-expr)

Then-statement

Else

Else-statement

Where test-expr is an integer expression that evaluates either to zero (interpreted as meaning “false”) or to a nonzero value (interpreted as meaning “true”). Only one of the two branch statements is executed.

For this general form, the assembly implementation typically adheres to the following form, where we use C syntax to describe the control flow:

t = test-expr;

if (!t)

goto false;

then-statement

goto done;

false:

else-statement

done;

That is, the compiler generated separate blocks of code for then-statement and else-statement. It inserts conditional and unconditional branches to make sure the correct block is executed.

3.6.6 Implementing Conditional Branches with Conditional Moves

The conventional way to implement conditional operations is through a conditional transfer of control, where the program follows one execution path when a condition holds and another when it does not. This mechanism is simple and general, but it can be very inefficient on modern processors.

An alternate strategy is through a conditional transfer of data. This approach computes both outcomes of a conditional operation and then selects one based on whether or not the condition holds. This strategy makes sense only in restricted cases, but it can then be implemented by a simple conditional move instruction that is better matched to the performance characteristics of modern processors. Here, we examine this strategy and its implementation with x86-64.

Figure 3.17(a) shows an example of code that can be compiled using a conditional move. The function computes the absolute value of its arguments x and y, as did our earlier example. Whereas the earlier example had side effects in the branches, modifying the value of either lt\_cnt or ge\_cnt, this version simple computes the value to be returned by the function.

To understand why code based on conditional data transfers can outperform code based on conditional control transfers (as in Figure3.16), we must understand something about how modern processors operate. As we will see in Chapters 4 and 5, processors achieve high performance through pipelining, where an instruction is processed via a sequence of stages, each performing one small portion of the required operations (e.g., fetching the instruction from memory, determining the instruction type, reading from memory, performing an arithmetic operation, writing to memory, and updating the program counter). This approach achieves high performance by overlapping the steps of the successive instructions, such as fetching one instruction while performing the arithmetic operations for a previous instruction. To do this requires being able to determine the sequence of instructions to be executed. When the machine encounters a conditional jump(referred to as a “branch”), it cannot determine which way the branch will go until it has evaluated the branch condition. Processors employ sophisticated branch prediction logic to try to guess whether or not each jump instruction will be followed. As long as it can guess reliably(modern microprocessor designs try to achieve success rates on the order of 90%), the instruction pipline will be kept full of instructions. Mispredicting a jump, on the other hand, requires that the processor discard much of the work it has already done on future instructions and then begin filling the pipeline with instructions starting at the correct location. As we will see, such a misprediction can incur a serious penalty, say, 15-30 clock cycles of wasted effort, causing a serious degradation of program performance.

As an example, we ran timings of the absdiff function on an Intel Haswell processor using both methods of implementing the conditional operation. In a typical application, the outcome of the test x < y is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. For X86-64 code with conditional jumps, we found that the function requires around 8 clock cycles per call when the branching pattern is easily predictable, and around 17.50 clock cycles per call when the branching pattern is random. From this, we can infer that the branch misprediction penalty is around 19 clock cycles. That means time required by the function ranges between around 8 and 27 cycles, depending on whether or not the branch is predicted correctly.

On the other hand, the code compiled using conditional moves requires around 8 clock cycles regardless of the data being tested. The flow of control does not depend on data, and this makes it easier for the processor to keep its pipeline full.

Figure3.18 illustrates some of the conditional move instructions available with x86-64. Each of these instructions has two operands: a source register or memory location S, and a destination register R. As with the different SET and jump instructions, the outcome of these instructions depends on the values of the condition codes. The source value is read from either memory or the source register, but it copied to the destination only if the specified condition holds.

The source and destination values can be 16, 32, or 64 bits long. Single condition move is not supported. Unlike the unconditional instructions, where the operand length is explicitly encoded in the instruction name, the assembler can infer the operand length of a conditional move instruction from the name of the destination register, and so the same instruction name can be used for all operand lengths.

**Unlike conditional jumps, the processor can execute conditional move instructions without having to predict the outcome of the test.** The processor simply reads the source value(possibly from memory), checks the condition code, and then either updates the destination register or keeps it the same.

To understand how conditional operations can be implemented via conditional data transfers, consider the following general form of conditional expression and assignment:

Figure 3.18 The conditional move instructions. These instructions copy the source value S to its destination R when the move condition holds. Some instructions have “synonyms”, alternate names for the same machine instruction.

V = test-expr ? then-expr : else-expr;

The standard way to compile this expression using conditional control transfer would have the following form:

if (!test-expr)

goto false;

v = then – expr;

goto done;

false:

v = else-expr;

done:

This code contains two code sequences—one evaluating then-expr and one evaluating else-expr. A combination of conditional and unconditional jumps is used to ensure that just one of the sequences is evaluated.

For the code based on a conditional move, both the then-expr and the else-expr are evaluated, with the final value chosen based on the evaluation test-expr. This can be described by the following abstract code:

v = then-expr;

ve = else-expr;

t = test-expr;

if (!t) v = ve;

The final statement in this sequence is implemented with a conditional move—value ve is copied to v only if test condition t does not hold.

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code we have shown evaluates both then-expr and else-expr regardless of the test outcome. If one of those two expressions could possibly generate an error condition or a side effect, this could lead to invalid behavior. Such is the case for our earlier example (Figure 3.16). Indeed, **we put the side effects into this example specifically to force GCC to implement this function using conditional transfers.**

As a second illustration, consider the following C function:

long created(long \*xp) {

return (xp ? \*xp :0);

}

At first, this seems like a good candidate to compile using conditional move to set the result to zero when the pointer is null, as shown in the following assembly code:

This implementation is invalid, however, since the dereferencing of xp by the movq instruction occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

Using conditional moves also does not always improve code efficiency. For example, if either the then-expr or the else-expr evaluation requires a significant computation, then this effort is wasted when the corresponding condition does not hold. Compilers must take into account the relative performance of wasted computation versus the potential for performance penalty due to branch misprediction. In truth, they do not really know how well the branches will follow predictable patterns. Our experiments with GCC indicate that it only uses conditional moves when the two expressions can be computed very easily, for example, with single add instructions. In our experience, GCC uses conditional control transfers even in many cases where the cost of branch misprediction would exceed even more complex computations.

Overall, then, we see that conditional data transfer offer an alternative strategy to conditional control transfers for implementing conditional operations. They can only be used in restricted cases, but these cases are fairly common and provide a much better match to the operation of modern processors.

3.6.7 Loops

C provides several looping constructs-namely, do-while, while and for. No corresponding instructions exist in machine code. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Gcc and other compilers generate loop code based on the two basic loop patterns. We will study the translation of loops as a progression, starting with do-while and then working toward ones with more complex implementations, covering both patterns.

Do-While Loops

The general form of a do-while statement is as follows:

do

body-statement

while (test-expr);

The effect of the loop is to repeatedly execute body-statement, evaluate test-expr, and continue the loop if the evaluation result is nonzero. Observe that body-statement is executed at least once.

This general form can be translated into conditionals and goto statements as follows:

loop:

body-statement

t = test-expr;

if (t)

goto loop;

That is, on each iteration the program evaluates the body statement and then the test expression. If the test succeeds, the program goes back for another iteration.

Aside Reverse engineering loops

A key to understanding how the generated assembly code relates to the original source code is to find a mapping between program values and registers. This task was simple enough for the loop of Figure3.19, but it can be much more challenging for more complex programs. The C compiler will often rearrange the computations, so that some variables in the C code have no counterpart in the machine code, and new values are introduced into the machine code that do not exist in the source code. Moreover, it will often try to minimize register usage by mapping multiple program values onto a single register.

(The process we described for fact\_do works as a general strategy for reverse engineering loops. Look at how registers are initialized before the loop, updated and tested within the loop, and used after the loop. Each of these provides a clue that can be combined to solve a puzzle. Be prepared for surprising transformations, some of which are clearly cases where the compiler was able to optimize the code, and others where it is hard to explain why the compiler chose that particular strategy.)

The conditional jump instruction jg is the key instruction in implementing a loop. It determines whether to continue iterating or to exit the loop.

Reverse engineering assembly code, such as that of Figure 3.19(c), requires determining which registers are used for which program values. In this case, the mapping is fairly simple to determine: We know that n will be passed to the function in register %rdi. We can see register %rax getting initialized to 1. We can see that this register is also updated by multiplication on line4. Furthermore, since %rax is used to return the function value, it is often chosen to hold program values that are returned. We therefore conclude that %rax corresponds to program value result.

While Loops

The general form of a while statement is as follows:

While (test-expr)

Body-statement

It differs from do-while in that test-expr is evaluated and the loop is potentially terminated before the first execution of body-statement. There are a number of ways to translate a while loop into machine code, two of which are used in code generated by GCC. Both use the same loop structure as we saw for do-while loops but differ in how to implement the initial test.

The first translation method, which we refer to as jump to middle, performs the initial test by performing an unconditional jump to the test at the end of the loop. It can be expressed by the following template for translating from the general while loop from to goto code:

goto test:

loop:

body-statement

test:

t = test-expr;

if (t)

goto loop;

The second translation method, which we refer to as guarded do, first transforms the code into a do-while loop by using a conditional branch to skip over the loop if the initial test fails. **GCC follows this strategy when compiling with higher levels of optimization**, for example, with command-line option -O1. This method can be expressed by the following template for translating from the general while loop form to a do-while loop:

t = test-expr

if (!t)

goto done;

do

body-statement

done:

This, in turn, can be transformed into goto code as

t = test-expr;

if(!t)

goto done;

loop:

body-statement

t = test-expr;

if (t)

goto loop;

done:

Using this implementation strategy, the compiler can often optimize the initial test, for example, determining that the test condition will always hold.

3.6.8 Switch Statements

A switch statement provides a multiway branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, but they also allow an efficient implementation using a data structure called a jump table. A jump table is an array where entry i is the **address** of a code segment implementing the action the program should take when the switch index equals i. The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a switch statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases and they span a small range of values.

Figure 3.23 shows the assembly code generated when compiling switch\_eg. The behavior of this code is shown in C as the procedure switch\_eg\_impl in Figure 3.22(b). This code makes use of support provided by GCC for jump tables, as an extension to the C language. The array jt contains seven entries, each of which is the address of a block of code. These locations are defined by labels in the code and indicated in the entries in jt by code pointers, consisting of the labels prefixed by &&. (Recall that the operator ‘&’ creates a pointer for a data value. In making this extension, the authors of GCC created a new operator && to create a pointer for a code location.) we recommend that you study the C procedure switch\_eg\_impl and how it relates to the assembly-code version.

Our original C code has cases for values 100, 102-104, and 106, but the switch variable n can be an arbitrary integer. The compiler first shifts the range to between 0 and 6 by subtracting 100 from n, creating a new program variable that we call index as an unsigned value, making use of the fact that negative numbers in a two’s-complement representation map to large positive numbers in an unsigned representation. It can therefore test