Branch Removal - A Pattern For Code Efficiency

Joseph W. Seda

17 October, 1997

Introduction

With CPU clock speeds skyrocketing, chip vendors are relying more and more on pipelining to increase performance. Pipelines improve speed by breaking up single instructions into smaller pieces that can be executed much more quickly. The 8088 had a 4 instruction prefetch and a 2 stage pipeline consisting of a Bus Interface Unit and an Execution Unit [1]. Newer chips have pipelines with 6 or more stages. This trend is sure to continue.

Branches, i.e. jumps caused by loops and conditional statements in the software, are a bane of pipelined CPU performance because they can cause delays. The CPU cannot know a-priori whether a branch will be taken or not, so it must guess and fetch instructions in the direction of its predicted path. If the guess is correct, all is well. If not, all the fetched instructions are wrong and must be flushed from the pipeline, causing delay.

Hardware mechanisms do exist which attempt to alleviate the problem by predicting the direction of the taken branch. Three examples are the *delayed branch*[2] instruction, the *branch target cache*[3] and *branch history*[4] . Other hardware techniques are being researched as well.[5] However these are not available on all platforms and their efficiency is highly dependent on runtime data. Compilers can also do some branch optimizations, but that ability varies widely across vendors. Although the author belongs to the camp which believes that performance should primarily come from hardware, performance will always be an issue in software. As Jim Coplien has said, "Real programmers can't ignore performance or efficiency; they are not academic concerns. Since patterns are about the 'real stuff' that we grapple with as designers and programmers, performance is fertile turf for pattern mining." [6]; This article describes a performance pattern to remove branches to achieve speedup in code running on pipelined hardware.

Pattern Name

Branch Removal

Problem

Branches caused by conditional statements, e.g. if() statements, slow CPU performance on pipelined hardware.

Context

CPU bound processing, 80/20 rule already applied, code nearly optimized. The critical section contains many branches involving integral types. Speed is a high priority, but assembly language is not on option. Code must be portable. 2’s complement instruction set. (1’s complement instruction sets, e.g. some mainframes, requires slight adjustment)

Forces

The code should be fast yet clear, maintainable, and testable. Branch removal causes code obfuscation. Compiler optimization quality decays over time. [7] Current compiler optimizers do not typically remove branches. Untransformed code may be faster, depending on data. Portability is vital.

Solution

Using the identity (if A then B) <==> not(A) OR B, replace conditional statements with bit manipulation in the sign bit.

Consequences

- (+) Execution time is lower on average.
- (+) Execution time is deterministic.
- (-) (Cyclomatic) Complexity is masked, causing code to be harder to understand.
- (-) Program flow is hidden.
- (-) Some register allocation optimizations may be thwarted and typically more instructions are required.
- (-) Branch coverage, a metric for testing, is rendered useless.
- (-/+) Code has the flavor of microcode.

Known Uses

The author was involved in a project involving ORDB technology at a benefits consulting company. A Numeric (fixed point decimal) class was needed to provide 18 digits of precision. The branch removal technique was used to eliminate a branch involved in the carry calculation in the Numeric's add() method, the hot spot. Depending on the calculation, from 5 to 15% speedup was obtained.

A computer game program, for example a program that plays checkers against a human, requires speed to play well. Often the bottleneck in such a program is in the search code that determines legal moves, which typically has deeply nested branches. The author has achieved up to 25% speedup for various game programs using the branch removal technique.

Example: Eliminating Nested Branches

Given the following code example, here are steps to eliminate a nested branch:

```
if( a < b )
```

```
    if( c < d )
         statement;
```

- Replace comparisons with subtractions. a < b becomes a - b, a <= b becomes a - b - 1, a > b becomes b - a, a >= b becomes b - a - 1. Here is the reasoning: if a < b, a - b will leave the sign bit 1 (negative), otherwise the sign bit will be 0.
- Replace nested if's with compound conditions && and ||.
- Flatten the compound conditions && and || with bitwise logic & and |. For example, a < b && c < d becomes (a - b) & (c - d).
- Use a mask to check the sign bit, which will be 1 if the condition is true and 0 if the condition is false. For an int, the value (0x1 << (8 * sizeof(int) - 1)) will work. This would be 0x8000 for 16 bit ints. Here is a useful macro which returns 1 if the sign bit is 1 and 0 otherwise: MSB(x) = (0x1 & ((x) >> (8 * sizeof(int) - 1))); This macro can be written as an inline function if you know that it will be inlined.

The code is now:

```
#define     MSB( x )        ( 0x1 &( (x) >> ( 8 * sizeof( int ) - 1 ) ) )

if( MSB( ( a - b  ) & ( c - d  ) ) )
    statement;
```

Example: Setting Values Based on a Condition

Using the following code example, here are steps to remove a branch used to set a value:

```
if( a < b )
    val = 23;
else
    val = 30;
```

- Using the steps from case 1, set the condition such that the sign bit is 1 if the condition is true.
- Create an array of 2 elements containing values to be set for the false and true conditions respectively.
- Set the value from the array indexed by the transformed condition.
- Here is useful function which returns 0 if a value is zero and 1 if the value is not.

  ```
  inline    int truthOf( A ) { return MSB( A ^= -A ); }
  ```

  Note that this function DOES NOT WORK for the special value of all zeroes with only the sign bit set. (sometimes called negative zero.) Use this function with care. An alternative expression, !!A, does not suffer the flaw, does not require a branch, and typically is faster and smaller. However, it is not strictly guaranteed to be portable.

The resulting code is

```
#define MSB( x )    ( 0x1 & ( (x) >> (8*sizeof(int)-1)) )
const static int    array[] ={ 30, 23 };
int val = array[ MSB( a - b ) ];
```

Example: Range Check

Compare the two versions of code in Listing 1, which shows a slightly less trivial example of branch removal. Using one compiler, the first version generates 26 assembly instructions and 4 branches. The second version generates 34 assembly instructions and 0 branches and runs faster on average. Note that this example is typical; removing branches may increase the number of instructions. If the number of instructions is too great, the code with branches may actually be quicker and the branch removal gives no benefit. In general if a CPU has N stages in its pipeline, branchless code requiring less than N instructions will be faster than the worst case for the branch.

```
//
//  UNTRANSFORMED
//  if 0 < x < MAX_X and 0 < y < MAX_Y
//      return true else return false
//
inline int      valid( int x, int y )
{
    if( x < 0 || y < 0 )
        return 0;
    if( x > MAX_X  )
        return 0;
    if( y > MAX_Y )
        return 0;
    return 1;
}


//
//  TRANSFORMED
//
inline int      valid( int x, int y )
{
    static int SHIFT_SIZE = 8 * sizeof( int ) - 1;
    static int SIGN_BIT = 0x1 << SHIFT_SIZE;

    return ( ( x | y | ( MAX_X - x ) |
        ( MAX_Y - y ) ) & SIGN_BIT ^ SIGN_BIT;
}
```

Listing 1.

Example: Triangle Problem

There are several drawbacks of the branch removal pattern. Removing branches hides the program flow. Since complexity is most often measured by the control statements, i.e. branches, the complexity is obscured. Another problem is that structural test completeness is typically measured by branch coverage[8]. With the branches removed, it can be very difficult to determine a comfortable level of testing.

The examples in Listing 2 and 3 demonstrate some shortcomings of branch removal. Both versions of the code implement the classic 'triangle' testing problem. This program accepts 3 integers as inputs which specify the length of the sides of a triangle. If the sides represent an equilateral triange, the output is "Triangle is Equilateral". If the sides represent a scalene triangle, the output is "Triangle is Scalene". If the sides represent an isosceles triangle, the output is "Triangle is Isosceles". If the sides do not form a triangle, the output is "Not a triangle".

Since the two programs produce the same results, they are equally complex. However, the second implementation is much more difficult to understand. Also note that because the second implementation has no branches, any one test case completely covers the code. Clearly one test case is not an adequate test; a metric other than branch coverage must be used to determine structural test completion.

```
//
//      Before Branch removal
//
#include        <iostream.h>
#include        <strstream.h>
#include        <fstream.h>
#include        <bool.h>

int     main(  )
{
    int a, b, c;
    char        buf[ 1024 ];

    bool        isATriangle = false;

    cin.getline( buf, sizeof( buf ) );
    istrstream  istr( buf, sizeof( buf ) );

    istr >> a >> b >> c;

    if( ( a < b + c ) && ( b < a + c )
            && ( c < a + b ) )
        isATriangle = true;

    if( isATriangle )
    {
        if( ( a == b ) && ( a == c ) )
            cout << "Triangle is Equilateral";
        else if( ( a != b ) && ( a != c ) && ( b != c ) )
            cout << "Triangle is Scalene";
        else
            cout << "Triangle is Isosceles";
    }
    else
        cout << "Not a Triangle";

    cout << endl;
    return      0;
}
```

Listing 2.

```
//
//      After Branch removal
//
#include        <iostream.h>
#include        <strstream.h>
#include        <fstream.h>

char*   existenceText[  ] = { "Not a Triangle\n", "" };
char*   isoscelesText[  ] = { "", "Triangle is Isosceles\n" };
char*   equilateralText[  ] = { "", "Triangle is Equilateral\n" };
char*   scaleneText[  ] = { "", "Triangle is Scalene\n" };
```

```
const static int SHIFT_SIZE = 8 * sizeof( int ) - 1;
#define MSB(n) ( 0x1 & ( (n) >> SHIFT_SIZE ) )

inline int truthOf( int val )
{
    val ^= -val;
    return      MSB( val );
}

int     main(  )
{
    int a, b, c;
    char        buf[ 1024 ];

    cin.getline( buf, sizeof( buf ) );
    istrstream  istr( buf, sizeof( buf ) );

    istr >> a >> b >> c;

    int oneGood = MSB( b + c - a - 1 ) ^ 0x1;
    int twoGood = MSB( a + c - b - 1 ) ^ 0x1;
    int threeGood = MSB( a + b - c - 1 ) ^ 0x1;
    int isATriangle = oneGood & twoGood & threeGood;

    int aNotEqualB = truthOf( a ^ b );
    int bNotEqualC = truthOf( b ^ c );
    int aNotEqualC = truthOf( a ^ c );

    int aEqualsB = aNotEqualB ^ 0x1;
    int bEqualsC = bNotEqualC ^ 0x1;
    int aEqualsC = aNotEqualC ^ 0x1;

    cout << existenceText[ oneGood & twoGood & threeGood ];
    cout << isoscelesText[ isATriangle &
                ( aEqualsB ^ aEqualsC ^ bEqualsC ) &
                ( aNotEqualB | aNotEqualC ) ];
    cout << equilateralText[ isATriangle & aEqualsB & bEqualsC ];
    cout << scaleneText[ isATriangle & aNotEqualB &
                                aNotEqualC & bNotEqualC ];

    cout.flush();
    return      0;
}
```

Listing 3.

Summary

Branch removal can provide a nearly portable alternative to assembly when speed is of the utmost priority and writing assembly is not an option. It can improve code speed on pipelined hardware, but diminishes readability, maintainability, and testability. Although the examples are in C++, any language which allows bit manipulation of integral values can use this technique. Thanks to Brad Appleton for his early comments and to Mike Beedle, Bob Hanmer, Jim Coplien and all the TCPG members for making me the first writer's workshop guinea pig there. Thanks also to John Spoelstra for the quality perspective and Robert Martin for the !!A expression.

References

- [1] Protopapas, D. A. "Microcomputer Hardware Design". Prentice-Hall Inc., Englewood Cliffs, New Jersey. p. 105
- [2] Stone, Harold S. "High-Performance Computer Architecture". Addison- Wesley, Reading Massachusetts. p. 150
- [3] Ibid.
- [4] Ibid.
- [5] Uht, Sindagi, Somanathan. "Branch Effect Reduction Techniques". IEEE Computer, May 1997
- [6] Coplien, James. "After All, We Can't Ignore Efficiency" C++ Report, May 1996
- [7] "Folklore says that compiler optimizations disappear gradually over time. Code generation bugs are often found long after the compiler writer has moved on. The easiest solution, and therefore the one most often applied, is disabling selected compiler optimizations." Thomas Christopher (personal communication)
- [8] Jorgenson, Paul C. "Software Testing - A Craftsman's Approach" CRC Press, Boca Raton, Florida. p. 139