JAN 13TH, 2013 12:00 AM

From Switch Statement Down to Machine Code

Introduction

Most of us know what a *switch* statement is and have probably been using it very often. No wonder why — switch statement is simple yet extremely expressive. It allows keeping the code compact while describing complex control flow. Putting the syntactic sugar aside, most developers also believe that using a switch statement results in a lot better, faster code. Not many knows if that is really true and why. The most common speculations supporting superior switch statement performance are:

- Compilers implement switch as a jump table and it is faster than an average number of conditional branches that the code would have taken otherwise. Therefore, the code generated from switch expressions is executed faster than if-then-else.

Recent Posts

C++ Devirtualization

Coding While Hungover

Smart Command Line Options Parsing

How Constant Is a Constant?

Why C++ Member Function Pointers Are 16 Bytes Wide

C++ Exceptions, Stack Trace and GDB Automation

Ask Smart Questions and You Will Succeed

- Compilers generate a binary lookup table to match the input value when using switch statement. The binary search algorithm's worst performance is O(log n). It is a faster than O(n) worst case performance of a linear search. Therefore, switch is faster.

  It all sounds good in theory. Yet not many understand switch statements down to the machine code level and even less have checked what their compilers do. By the way, why cannot a compiler optimize an if-then-else code the same way it optimizes a switch? Maybe it can? Let's find out!

  We'll use the two most popular production quality compilers — GCC (version 4.7.2, released 20 Sep 2012) and Clang (version 3.0, released December 01, 2011).

  In both cases, we will be compiling the code for Intel® Xeon® E5630 CPU with enabled compiler optimizations (either «-O2», «-O3» or «-Os»).

  The knowledge of x86_64 assembly language is not required but some understanding of what it is and how CPU process low-level instructions might be helpful.

Simple Switch

Let's start by looking at a very simple switch statement. It has six case labels, does not have a default case, and has no fall-trough cases so each case is followed by a break statement:

```
1  #include <time.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main()
6  {
```

```
    srand(time(NULL));
    switch (rand()) {
    case 0:
        printf("zero\n");
        break;
    case 1:
        printf("one\n");
        break;
    case 2:
        printf("two\n");
        break;
    case 3:
        printf("three\n");
        break;
    case 4:
        printf("four\n");
        break;
    case 5:
        printf("five\n");
        break;
    }
    return EXIT_SUCCESS;
}
```

It would be logical to assume that the above code would be translated into a jump table. To verify this assumption, we need to disassemble the binary generated by the compiler and check how the machine code looks like.

Below is a disassembly of the binary generated by GCC compiler. Each line has a comment on the right that starts with `;` character:

```
4004f7:       cmp    eax,0x5                  ; Compare the value with 5.
4004fa:       ja     400518 <main+0x38>       ; Jump to exit if the value is greater t
```

```
4004fc:       mov    eax,eax                    ; Clear 32 to 63 bits
4004fe:       xchg   ax,ax                      ; NOP
400500:       jmp    QWORD PTR [rax*8+0x400738] ; Jump to address 0x400738[%rax]
400507:       mov    edi,0x40072c               ; Set "five" as a parameter.
40050c:       call   400490 <puts@plt>          ; Call "puts()".
400511:       nop    DWORD PTR [rax+0x0]        ; NOP
400518:       xor    eax,eax                    ; Zero return code.
40051a:       add    rsp,0x8                    ; Pop stack (stack grows inward).
40051e:       ret                               ; Return from the main() function.
40051f:       mov    edi,0x400727               ; Set "four" as a parameter.
400524:       call   400490 <puts@plt>          ; Call "puts()".
400529:       jmp    400518 <main+0x38>         ; Jump to exit.
40052b:       mov    edi,0x400714               ; Set "three" as a parameter.
400530:       call   400490 <puts@plt>          ; Call "puts()".
400535:       jmp    400518 <main+0x38>         ; Jump to exit.
400537:       mov    edi,0x400719               ; Set "two" as a parameter.
40053c:       call   400490 <puts@plt>          ; Call "puts()".
400541:       jmp    400518 <main+0x38>         ; Jump to exit.
400543:       mov    edi,0x40071d               ; Set "one" as a parameter.
400548:       call   400490 <puts@plt>          ; Call "puts()".
40054d:       nop    DWORD PTR [rax]            ; NOP
400550:       jmp    400518 <main+0x38>         ; Jump to exit.
400552:       mov    edi,0x400721               ; Set "zero" as a parameter.
400557:       call   400490 <puts@plt>          ; Call "puts()".
40055c:       jmp    400518 <main+0x38>         ; Jump to exit.
40055e:       xchg   ax,ax                      ; Unreachable NOP (padding).
```

As we can see, GCC has generated the code that uses an indirect jump transferring execution of the program to a code location depending on the input value using a combination of cmp and ja instructions. This is a simple case of a jump table.

To see what other compilers might do, let's look at the disassembly of the same program compiler by Clang:

```
4005a7: cmp    eax,0x5                    ; Compare the value with 5.
4005aa: ja     4005e2 <main+0x52>         ; Jump to exit if value is > 5.
4005ac: mov    eax,eax                    ; Convert 32-bit to 64-bit (clear upper half)
4005ae: jmp    QWORD PTR [rax*8+0x4006e0] ; Jump to a relative address using %rax as th
4005b5: mov    edi,0x400710               ; Push "zero" as parameter.
4005ba: jmp    4005dd <main+0x4d>         ; Go to print & exit.
4005bc: mov    edi,0x400715               ; Push "one" as parameter.
4005c1: jmp    4005dd <main+0x4d>         ; Go to print & exit.
4005c3: mov    edi,0x400719               ; Push "two" as parameter.
4005c8: jmp    4005dd <main+0x4d>         ; Go to print & exit.
4005ca: mov    edi,0x40071d               ; Push "three" as parameter.
4005cf: jmp    4005dd <main+0x4d>         ; Go to print & exit.
4005d1: mov    edi,0x400723               ; Push "four" as parameter.
4005d6: jmp    4005dd <main+0x4d>         ; Go to print & exit.
4005d8: mov    edi,0x400728               ; Push "five" as parameter.
4005dd: call   400450 <puts@plt>          ; Call "puts()" (printing point).
4005e2: xor    eax,eax                    ; Zero return (exit point).
4005e4: pop    rbp                        ; Pop stack frame.
4005e5: ret                               ; Return
4005e6: nop    WORD PTR cs:[rax+rax*1+0x0] ; NOP (alignment/padding)
```

From the above example, we can immediately tell that Clang has generated a more compact code by avoiding multiple calls to puts() function.

Interestingly enough, GCC can also eliminate multiple calls to puts() function and generate exactly the same code for the above switch statement as Clang does. To achieve this, we must specify «-Os» flag on the command line. This flags instructs the compiler to optimize the binary for small size.

Apart from this small difference in optimizing function calls, both compilers used a combination of cmp and ja functions to implement a jump table. From this we can draw a conclusion that for simple cases compilers do generate a jump table.

Crafting a Jump Table

To get a better understanding of a jump table mechanics, let's consider creating one manually without using a switch statement. It is possible to create a jump table manually a few different ways.

The first way of writing a jump table is to use an assembler language. It is low level programming language that does not have high-level constructs such as if-then-else or switch, but it allows to use processor instructions directly, making it possible to manually employ "cmp", "ja" or other instructions to implement a functional equivalent of the code generated by compilers. This would have been very time consuming, require much higher understanding of the assembly language, result in less portable and hard to maintain code.

It is also possible to code a logical equivalent of a jump table in C, without using a switch statement. To do this, we could create an array that holds function pointers, and then call a function using an index to this array. For example:

```
1  #include <stdio.h>
2
3  static void zero(void) {
4      puts("zero");
5  }
6
7  static void one(void) {
8      puts("one");
9  }
10
11 static void two(void) {
12     puts("two");
13 }
14
```

```c
typedef void (*func)(void);

int main() {
    func table[] = { &zero, &one, &two };
    int i = 0;
    /* Read the value of `i` from user
       input or other source */
    table[i]();
}
```

This, however, would only be a logical equivalent of the jump table, but not a functional one. The main difference in functionality is that jump table transfers control to a specified location directly instead of calling a function. To achieve a functional equivalent of the jump table in C or C++, we must structure the code in such a way as to avoid function calls. We must create an array of addresses where the control is to be transferred, instead of array of functions. One of the possible ways of doing this is to label our code and create an array that store labels instead of functions. For example:

```c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
        static const void *jump_table[] = {
                &&print_0, &&print_1, &&print_2,
                &&print_3, &&print_4, &&print_5
        };
        int v;

        srand(time(NULL));
        v = rand();
        if (v < 0 || v > 5)
                goto out;
```

```
        goto *jump_table[v];

    print_0:
        printf("zero\n");
        goto out;
    print_1:
        printf("one\n");
        goto out;
    print_2:
        printf("two\n");
        goto out;
    print_3:
        printf("three\n");
        goto out;
    print_4:
        printf("four\n");
        goto out;
    print_5:
        printf("five\n");
        goto out;
out:
    return EXIT_SUCCESS;
}
```

The above code would result in exactly the same binary as produced by GCC for a simple switch statement example we have reviewed in the first chapter. It cannot be considered readable or easily maintainable, but it could have been used for performance reasons if C or C++ language did not have a switch statement construct.

Note that despite being used by low-level developers from time to time, the above code is not standard. It is neither standard C nor C++ is none of those languages support taking address of a label. Despite that, this feature is implemented as a nonstandard extension by most production grade C and C++ compilers.

Trivial Switch

Modern processors employ a wide variety of optimization techniques to speed up the execution of computer programs. One of those techniques is branch prediction. It tries to guess which way a branch of the logic will go before this is known for sure. In a high-level programming languages such as C, branches are formed by if-then-else structures, goto instructions, different kinds of loop statements, switch statement and other constructs.

How processors implement branch prediction is largely a trade secret. It is well known, however, that this technique is less efficient if processor runs into indirect branch instruction, such as that used in jump tables code. In other words, a single indirect jump instruction is relatively more expensive than a simple branch, such as generated by if-then construct.

This means that for some architectures, it is possible that a simple if-then-else statement with a few comparisons and direct jump instructions might theoretically execute faster than a single indirect jump instruction. In other words, if compilers always generate a jump table for a switch statement, then we could write a more efficient code (from execution time perspective) by avoiding using a switch and resorting to if-then-else.

The cost ratio of compare and jump instructions to indirect jump instruction is not known without experimentation. How much if-then-else branches can we take before its execution time would be slower than that of a single indirect jump? It is also not clear whether compilers take care of this or not. To find out, we must experiment. Since our initial switch statement had six case labels, we must keep reducing the number of labels, disassemble the resulting binary on every stage and compare the results to see if the generated code is functionally different.

Five Cases

The first step is to remove a single label from the switch, effectively reducing a number of case labels to five. By doing so, nothing has changed regarding how compilers handle switch statement in case with both GCC and Clang compilers.

Four Cases

Reducing a number of cases further results in some interesting changes. With only four cases in a switch, Clang continues to generate a jump table as in all previous cases. GCC, on the other hand, stops using a jump table and resorts to simple comparison equivalent to if-then-else. Below is a disassembled binary demonstrating what GCC does in this case:

```
4004f4:  cmp    eax,0x1              ; Compare value with 1.
4004f7:  je     400514 <main+0x34>   ; If value is 1, go there
4004f9:  jg     400501 <main+0x21>   ; If value is greater than 1, compare more.
4004fb:  test   eax,eax              ; Could be <= 0. Compare with 0.
4004fd:  je     40050d <main+0x2d>   ; Go ptint "zero" if matched or jump to exit
4004ff:  jmp    40052c <main+0x4c>   ; Jump to exit (value was not zero)
400501:  cmp    eax,0x2              ; Value was > 1. Compare it with 2.
400504:  je     40051b <main+0x3b>   ; Matched, go print and exit.
400506:  cmp    eax,0x3              ; Was != 2. Compare with 3.
400509:  jne    40052c <main+0x4c>   ; Go to exit if not matched.
40050b:  jmp    400522 <main+0x42>   ; Value is three!!! Go print it.
40050d:  mov    edi,0x4006e4         ; Set "zero" as parameter.
400512:  jmp    400527 <main+0x47>   ; Go call "puts()" and exit.
400514:  mov    edi,0x4006e9         ; End up here if value is 1.
400519:  jmp    400527 <main+0x47>   ; Go call "puts()" and exit.
40051b:  mov    edi,0x4006ed         ; Value was 2. Set...
400520:  jmp    400527 <main+0x47>   ; ... "two" as parameter and go print.
400522:  mov    edi,0x4006f1         ; Set "three" as argument to "puts()".
400527:  call   400490 <puts@plt>    ; Call "puts()".
40052c:  xor    eax,eax              ; Zero register with return value.
40052e:  pop    rdx                  ; Pop stack
40052f:  ret                         ; Return control to the caller.
```

As we can see, there is no longer an indirect jump. Instead, the input is compared with every possible value using a set of compare and jump instructions.

Three and Less Cases

With only three cases in a switch, Clang starts generating the same code as GCC does starting at four — comparison instructions are used instead of a jump table. By reducing a number of cases further down to one, the same result is observed with both compilers.

Note on GCC

GCC compiler allows users to control the cutoff between doing switch statements as a series of if-then-else statements and using a jump table since version 4.7. The threshold controlling this behavior can be specified as a command line option.

Conclusion

From the above experiments, we can conclude that both GCC and Clang compilers are well aware that indirect jumps are relatively expensive.

Both compilers are trying to avoid a jump table if the number of case labels in the switch is small enough to justify using a chain of compare and jump instructions.

The only difference between the two compilers is they use a different cost ratio when deciding on using a jump table. GCC drops the idea of jump table starting at 4 case labels down to 1. Clang drops the jump table approach at 3 swithc cases and below.

This of course is purely an implementation detail and can change from one platform to another, or between different versions of the compiler.

Default Case

Does having a switch statement with a default case affect the mechanism used to implement a switch statement in machine code?

Unlike hardware description languages like Verilog where there could be no default case in a switch, the software logic always has a default case that transfers a control flow further. It might be explicitly specified with a special default case label, or be implicitly generated by a compiler.

Therefore, the presence or absence of an explicit default case does not make a difference. The only case when not specifying a default case explicitly is beneficial is when switch is performed on enumeration. In that case, compiler may warn a programmer if switch does not handle all possible values of enumeration. This is strictly a static analysis feature that is helpful to developers but does not affect compiler's decision about the implementation mechanism of a switch statement in any way.

Large Values

Not all switch cases have their values starting at 0. Some may have them start at one, two or even a million. Since jump tables are essentially represented as arrays, and all arrays in both C and C++ languages start with 0. We certainly cannot create an array of 101 elements only to have the last entry at index 100 to hold a valid jump address. So how does having large values in the switch affect the generated code?

Continuous Range

Let's first take a look at a very simple example where all case values of a switch are continues:

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    srand(time(NULL));
    switch (rand()) {
      case 1986000:
          printf("zero\n");
          break;
      case 1986001:
          printf("one\n");
          break;
      case 1986002:
          printf("two\n");
          break;
      case 1986003:
          printf("three\n");
          break;
      case 1986004:
          printf("four\n");
          break;
      case 1986005:
          printf("five\n");
          break;
    }
    return EXIT_SUCCESS;
}
```

As you can see, the minimum value in the switch is 198,600,000. Some programmers think that, because compiler build a jump table from case values and the input to a switch statement might be used as an index to a jump table, having large values will not work and result in less efficient code. So what they are trying to help the compiler optimize the code by writing it like this:

```c
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    srand(time(NULL));
    switch (rand() - 1986000) {
      case 0:
          printf("zero\n");
          break;
      case 1:
          printf("one\n");
          break;
      case 2:
          printf("two\n");
          break;
      case 3:
          printf("three\n");
          break;
      case 4:
          printf("four\n");
          break;
      case 5:
          printf("five\n");
          break;
    }
    return EXIT_SUCCESS;
}
```

In other words, they subtract N from input value to make the resulting range start from 0. Is this a good optimization?

Both GCC and Clang are production grade compilers. It is extremely rare to run into a situation where programmer needs to optimize such a simple case manually, unless programmer knows something that compiler cannot possibly guess from context. Therefore, it would be logical to assume that both GCC and Clang should handle the above optimization automatically. To verify this, below is a disassembled binary code generated by GCC from the code without a manual optimization:

```
4004f2:       call   4004d0 <rand@plt> ; Call "rand()".
4004f7:       sub    eax,0x1e4dd0      ; Subtract "1986000" from the result.
4004fc:       cmp    eax,0x5           ; Compare with 5 and do the rest as before.
```

The code generated by the Clang looks exactly the same:

```
4005a2:       call   400490 <rand@plt> ; Call "rand()".
4005a7:       add    eax,0xffe1b230    ; Add "4292981296" to the result.
4005ac:       cmp    eax,0x5           ; Compare with 5 and do the rest as before.
```

This proves that compilers are smart enough to perform basic optimization on switch statements whose values are not starting with 0.

Sparse Values

So far we have looked at simple switch statements whose case values were continuous. They either started from 0 or other numbers and incremented continuously without gaps in between.

The Wikipedia article on switch statement quotes a research paper:

> *To optimize a switch statement, the programmer must use a very compact range of possible values to test.*

What happens if that is not possible or if programmer does something differently? Does it mean that a switch statement would not be optimized in any way?

There are two common types of values distribution in sparse switches. The first is when values can still be grouped together. For example, values of 10, 11, 12, 100 and 101 can grouped into two compact ranges, [10-12] and [100-101]. The second case is when no grouping can be done whatsoever.

Let's test those two cases and find out what optimizations are performed by the compiler, if any.

Distant Ranges

Consider the following switch statement with values in [0-5] and [10000-10002] ranges:

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    srand(time(NULL));
    switch (rand()) {
      case 0:
          printf("zero\n");
          break;
      case 1:
```

```
            printf("one\n");
            break;
        case 2:
            printf("two\n");
            break;
        case 3:
            printf("three\n");
            break;
        case 4:
            printf("four\n");
            break;
        case 5:
            printf("five\n");
            break;

        case 10000:
        case 10001:
        case 10002:
            printf("10K!\n");
            break;
    }
    return EXIT_SUCCESS;
}
```

Below is the disassembled binary code generated for the above code by GCC:

```
4004f7:   cmp    eax,0x3              ; Comparison #1: Compare with 3.
4004fa:   je     40056e <main+0x8e>   ; It is 3! Print & exit.
4004fc:   nop    DWORD PTR [rax+0x0]
400500:   jle    40051f <main+0x3f>   ; Is it < 3? Go to comparison #3.
400502:   cmp    eax,0x5              ; Comparison #2: Compare with 4.
400505:   je     400553 <main+0x73>   ; It is 4! Print & exit.
400507:   jl     400537 <main+0x57>   ; Is it < 4? Goto comparison #4.
400509:   sub    eax,0x2710           ; Subtract 10000.
40050e:   cmp    eax,0x2              ; Comparison #5: Compare with 2.
```

```
400511:   ja     400530 <main+0x50>  ; It is >2. Just exit (no match).
400513:   mov    edi,0x400741        ; Set "10K!" as a parameter
400518:   call   400490 <puts@plt>   ; Call "puts()".
40051d:   jmp    400530 <main+0x50>  ; Go to exit.
40051f:   cmp    eax,0x1             ; Comparison #3: Compare with 1.
400522:   je     400562 <main+0x82>  ; It is 1! Go print it.
400524:   jle    400543 <main+0x63>  ; It is less than 1. Go to comparison #6.
400526:   mov    edi,0x40072d        ; The value is 2 (by exclusion),
40052b:   call   400490 <puts@plt>   ; ... print "2" and exit.
400530:   xor    eax,eax
400532:   add    rsp,0x8
400536:   ret
400537:   mov    edi,0x400737
40053c:   call   400490 <puts@plt>
400541:   jmp    400530 <main+0x50>
400543:   test   eax,eax             ; Comparison #6
400545:   jne    400530 <main+0x50>  ; Nope. No match here. Go to exit.
400547:   mov    edi,0x400724        ; Print "zero" and exit.
```

As we can see, the generated code seems to be less efficient compared to that of jump table. By looking closer at the program flow we can see that GCC has implemented a binary search algorithm. It is still well optimized compared to a simple chain of compare and jump instructions generated by if-then-else.

Clang takes a different approach in handling the above code:

```
4005a7:   cmp    eax,0x270f
4005ac:   jg     4005e6 <main+0x56>
4005ae:   cmp    eax,0x5
4005b1:   ja     4005fb <main+0x6b>
4005b3:   mov    eax,eax
4005b5:   jmp    QWORD PTR [rax*8+0x4006f0]
4005bc:   mov    edi,0x400720
```

```
4005c1:   jmp    4005f6 <main+0x66>
4005c3:   mov    edi,0x400725
4005c8:   jmp    4005f6 <main+0x66>
4005ca:   mov    edi,0x400729
4005cf:   jmp    4005f6 <main+0x66>
4005d1:   mov    edi,0x40072d
4005d6:   jmp    4005f6 <main+0x66>
4005d8:   mov    edi,0x400733
4005dd:   jmp    4005f6 <main+0x66>
4005df:   mov    edi,0x400738
4005e4:   jmp    4005f6 <main+0x66>
4005e6:   lea    eax,[rax-0x2710]
4005ec:   cmp    eax,0x3
4005ef:   jae    4005fb <main+0x6b>
4005f1:   mov    edi,0x40073d
4005f6:   call   400450 <puts@plt>
4005fb:   xor    eax,eax
4005fd:   pop    rbp
4005fe:   ret
```

The code generated by Clang is a little bit more tricky compared to GCC. First it compares the input value with 9999. If the value is greater, it normalizes it by subtracting 10000 using “lea“ instruction, and then checks if the value is within [0-3) range. If it is, the “10K” is then printed. Otherwise, the function returns. If value is less than 9999, it ensures the value is in [0-5] range and uses a jump table. Very clever, isn’t it?

Sparse Values

The second common switch pattern is when values are sparse enough so that they cannot be grouped together. For example:

```
#include <time.h>
```

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    srand(time(NULL));
    switch (rand()) {
      case 0: printf("0!\n"); break;
      case 50: printf("50!\n"); break;
      case 100: printf("100!\n"); break;
      case 150: printf("150!\n"); break;
      case 200: printf("200!\n"); break;
      case 250: printf("250!\n"); break;
      case 300: printf("Spartans!\n"); break;
      case 350: printf("350!\n"); break;
      case 400: printf("400!\n"); break;
      case 450: printf("400!\n"); break;
      case 500: printf("400!\n"); break;

      case 10000:
      case 10001:
      case 10002:
      case 10003:
      case 10004:
      case 10005:
          printf("10K!\n");
          break;
    }
    return EXIT_SUCCESS;
}
```

For the above code, both GCC and Clang compilers have generated a binary search algorithm. Changing the order of case labels in the switch did not affect the generated code — both compilers have analyzed all the specified values before making optimization decisions.

Fall Through Switch

So far we have looked at switch statements with no fall through cases, where each case had a trivial code. There are of course more complex scenarios.

What happens if we introduce fall through switch? What if we make each case statement to have complex control flows? What if we do both of the above?

The answer is simple — it does not fundamentally change the way compilers implement switch statements. The same approach of mixing comparison, range checking, binary search and jump table logic is still used.

What it affects, however, is how compiler rearranges the control flow. It may group some logic together, or split it. Redundant code might be reduced, or it might get duplicated intentionally to reduce a number of jump instructions. The code might be placed at different addresses, compiler might also generate different jump instructions to achieve the most compact and fast code. All those optimizations techniques are not unique to switch statements and are being used with other language constructs.

Outsmarting Compilers

Every discussion about code micro-optimization, which switch statements are usually part of, shall start with a word of warning. Outsmarting a production quality compiler these days is a nearly impossible task. A programmer should not even try to optimize the code that is not proven to be a bottleneck by carefully profiling the whole program. If a piece of code is proven to be slow and there is an obvious optimization that compiler has failed to perform, shop for a better compiler. Start by

optimizing the logic and not the code — doing less steps where possible, avoiding chaotic dynamic memory manipulations, using well designed data structures will pay off more than any micro-optimization.

If, however, a switch statement turns out to be one of the biggest bottlenecks of the code, there might be ways to improve it. To do that, it is vital to know more information about the possible values passed into a switch than compiler knows about or may figure out from context. If there is nothing known about the input value — don't bother optimizing, there is no way of implementing a general case better than a compiler.

Improving a Switch

Consider the following example:

```
switch (value) {
  case 0:
      do_very_important_stuff_0();
      break;
  case 1:
      do_not_important_stuff_1();
      break;
  case 2:
      do_not_important_stuff_2();
      break;
  case 3:
      do_not_important_stuff_3();
      break;
  case 4:
      do_not_important_stuff_4();
      break;
  case 5:
      do_very_important_stuff_5();
```

```
        break;
}
```

The above switch statement will be implemented using a jump table. It is impossible to do better unless it is known that most of the time the value is either 0 or 5, and can turn out to be 1, 2 or 3 only in some rare conditions that almost never happen. If that is the case, there is no way that compiler may know about it (unless of course we use profiler feedback optimization, which is not always possible). So how to optimize this for that particular case?

We remember that an indirect call is more expensive than two-three comparisons, at least for our given platform. This was proven during our earlier experiments with reducing a number of cases in a switch. Given that our critical path in the above examples consists of only two values — 0 and 5, the code can be optimized by not using those two statements in a switch case to avoid an expensive indirect jump. For example:

```
if (value == 0) {
    do_very_important_stuff_0();
} else if (value == 5) {
    do_very_important_stuff_5();
} else {
    switch (value) {
      case 1:
          do_not_important_stuff_1();
          break;
      case 2:
          do_not_important_stuff_2();
          break;
      case 3:
          do_not_important_stuff_3();
          break;
      case 4:
          do_not_important_stuff_4();
```

```
            break;
        }
    }
```

If there are only two or three possible input values, the same technique can be used to improve switch statements that compiler implements using a binary search approach. If, however, the input value may vary, improving a switch performance turns into nothing more but improving a search algorithm. The programmer will have to analyze the most common set of input values, select or invent the search algorithm with better average performance or better performance for more important inputs, and manually implement the switch functionality using handcrafted if-then-else statements, creating a jump table manually (as described in chapter 3), or both.

Switch vs High-Level Search

Some developers occasionally get concerned whether it is better to use a switch statement or a higher-level search algorithms like, for example, a hash lookup implemented by the std::unordered_map class from C++ Standard Library.

Hash Lookup

Compared to dynamic hash lookup tables, statically generated lookup code for a pre-defined ranges of values will always be better than any other hash table implementation given that switch statements can only work with constant simple numeric POD types.

Binary Search

Ordered lookup algorithms such as those commonly used with std::map can theoretically be more efficient than a switch statement. The runtime overhead of those algorithms, however, may render them useless compared to a simple, low-level switch implementation. This may also depend on a

nature of the input. Therefore, there is no general answer to this question and developers must test both implementations to determine which one is better in any particular case.

If-then-else Recognition

By experimenting with different switch statements we have ensured that compilers do a wonderful optimization job. It almost does not matter how programmers describe the switch it in the code — compiler will always generate nearly perfect generic lookup algorithms, rearrange code as needed, group or duplicate statements and apply other techniques to make the code most efficient.

C programming language is very high-level and it might seem that in many cases it should not matter how the control flow is described and the same optimization might be applied as long as a resulting program is functionally the same. This raises a question of whether compiler will do the same optimizations if programmer is using if-then-else. In theory, it might be the same, or might be different. But instead of guessing, let's find out.

Below is a C program that is equivalent to our simple switch example:

```
1  #include <time.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main()
6  {
7      int v;
8
9      srand(time(NULL));
10     v = rand();
11
```

```
    if (v == 0) {
        printf("zero\n");
    } else if (v == 1) {
        printf("one\n");
    } else if (v == 2) {
        printf("two\n");
    } else if (v == 3) {
        printf("three\n");
    } else if (v == 4) {
        printf("four\n");
    } else if (v == 5) {
        printf("five\n");
    }

    return EXIT_SUCCESS;
}
```

GCC did not generate an equivalent code to that of a switch statement. A set of compare instructions is used instead. Here is a relevant snippet of the binary code:

```
4004f7:   test   eax,eax
4004f9:   je     40051e <main+0x3e>
4004fb:   cmp    eax,0x1
4004fe:   xchg   ax,ax
400500:   je     400536 <main+0x56>
400502:   cmp    eax,0x2
400505:   je     400542 <main+0x62>
400507:   cmp    eax,0x3
40050a:   je     40054e <main+0x6e>
40050c:   cmp    eax,0x4
40050f:   nop
400510:   je     40055a <main+0x7a>
400512:   cmp    eax,0x5
400515:   je     40052a <main+0x4a>
```

Clang compiler, on the other hand, has generated a jump table:

```
4005a7:   cmp    eax,0x5
4005aa:   ja     4005e2 <main+0x52>
4005ae:   jmp    QWORD PTR [rax*8+0x4006e0]
```

Therefore, we can draw a conclusion that compilers can and sometimes do optimize if-then-else statements the same way they optimize switch statements.

Whether it makes sense or not is another question. At first, it may seem like Clang has done a lot better job than GCC. However, this automatically prevents developer from performing optimizations described in the previous chapter by making it impossible to use comparison in the fast path of the program. This also prevents developers from manually providing branch prediction hints because switch cases, unlike if-then-else branches, cannot be explicitly prioritized. Since compilers cannot know for sure if switch was replaced by if-then-else statement on purpose, automatically replacing if-then-else with switch might discard programmer's optimization efforts and worsen the runtime efficiency of the program.

Summary

We have learned how decent compilers transform higher-level switch statements into a low level machine code, reviewed a number of different examples along with optimization techniques applied by the compilers.

We also discussed a few optimization techniques that can be applied in certain situations to improve the application performance by using a special mix of switch and if-then-else statements.

I hope that it would helps us, developers, to better understand what switch statements are, what they can be used for and how they work, as well as adding some practical backup to some very common speculative talks about switch statement optimizations.

Posted by Vlad Lazarenko • Jan 13th, 2013 12:00 am • tech

Tweet

« FPGA Programming. Where to begin?

Erasing Vector The Smart Way »