Complexity of `switch()` construct in gcc

Ivan Voras <ivoras@gmail.com>

1. Introduction

The `switch` construct in C has the mixed blessing of being much used and abused throughout the history of the language. The most famous of abuses is certainly the "Duff's device" which manages to interleave `switch..case` construct with a `for` construct to achieve loop unrolling.

One interesting property of this construct is that sometimes it's possible to implement it in machine code with O(1) complexity (with respect to the number of separate cases in the `switch..case` construct). This is possible mostly because of the main limitation of the construct – that it only works for integer and related type variables.

1.1. The environment

All code presented in this document is written and compiled on `gcc 3.4.4` on FreeBSD on i386. Unless otherwise noted, code is generated with `-march=pentium3`.

1.2. Basic code snippet

Code examined in this document is generally similar to this one:

```
enum t_bla { bla0, bla1, bla2, bla3, bla4 };

int main(int argc) {
    char bla = argc-1;

    switch (bla) {
        case bla0:
            printf("0\n");
            break;
```

```
        case bla1:
            printf("1\n");
            break;
        case bla2:
            printf("2\n");
            break;
        case bla3:
            printf("3\n");
            break;
        case bla4:
            printf("4\n");
            break;
        default:
            printf("default\n");
    }

    return 0;
}
```

The `enum` line is here in an attempt to fool the compiler, as is the first line in `main()`. The rest is a straightforward `switch..case` construct.

2. Threshold of optimisation

It turns out that the O(1) optimisation isn't done mindlessly, but with a threshold. If the code contains up to and including three case parts (excluding the default part), the generated code will be more or less straightforward:

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movzbl  8(%ebp), %eax
        andl    $-16, %esp
        subl    $16, %esp
        decb    %al
        movsbl  %al,%eax
        cmpl    $1, %eax
        je      .L4
        jle     .L10
        cmpl    $2, %eax
        je      .L11
.L6:
        movl    $.LC3, (%esp)
.L8:
        call    puts
        leave
        xorl    %eax, %eax
```

```
        ret
        .p2align 4,,7
.L10:
        testl   %eax, %eax
        jne     .L6
        movl    $.LC0, (%esp)
        jmp     .L8
        .p2align 4,,7
.L4:
        movl    $.LC1, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L11:
        movl    $.LC2, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
```

This assembler snippet is generated with `-O3` optimisation switch. Code corresponding to the `switch..case` construct is emphasized in bold letters in the above snippet. The `switch..case` construct from which the above code is generated has four parts, `case 0`, `case 1`, `case 2`, and the `default` case. Here are some observations about the generated code:

- Code is out of order. Compiler took the middle value as a anchor-case and did sort of binary partition on it.
- The comparison is always done with full 32-bit registers (the i386 architecture can also access lower 8 or 16 bits of general-purpose registers)
- The compiler obviously took advantage of the fact that there's only a limited (and well-known) set of possible values in the code
- There are two compare instructions generated, and due to other optimisations performed, the function ended up with three exit points (return instructions).

This is pretty smart code generation. There is out of order semantics, some fall-through optimisations and all calls to `printf()` were replaced with `puts()` because the printed strings didn't contain formatting specifiers. Also interesting to note is that generated code for the `switch..case` construct is really short, compared to the book-keeping code preceding it.

The generated code remains the same even if the `case X` values are out of order and/or non-sequential. Because all values are unique integers there's always the implicit ordering of numbers that is taken advantage of.

3. The O(1) case

With four or more `case` parts in the construct, and regardless of the compiler's optimisation switch (`-Ox`) a different algorithm is used, one which generates a jump table of addresses and uses the argument in `switch()` as the index into this table. The original C code presented at the start of this document generates this code (again, this is with `-O3` optimisation):

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movzbl  8(%ebp), %eax
        andl    $-16, %esp
        subl    $16, %esp
        decb    %al
        movsbl  %al,%eax
        cmpl    $4, %eax
        ja      .L8
        jmp     *.L9(,%eax,4)
        .section        .rodata
        .p2align 2
        .p2align 2
.L9:
        .long   .L3
        .long   .L4
        .long   .L5
        .long   .L6
        .long   .L7
        .text
        .p2align 4,,7
.L8:
        movl    $.LC5, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L7:
        movl    $.LC4, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L3:
        movl    $.LC0, (%esp)
```

```
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L4:
        movl    $.LC1, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L5:
        movl    $.LC2, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
        .p2align 4,,7
.L6:
        movl    $.LC3, (%esp)
        call    puts
        leave
        xorl    %eax, %eax
        ret
```

Again, the code corresponding to switch..case construct has been highlighted by bold using letters. After the check for the default value (`cmpl $4, %eax; ja .L8`) the compiler has embedded a small data section into the code. Some observations:

- The argument to the `switch()` is used directly to calculate the index into the jump table with one machine instruction.
- Each of the cases have been transformed into an exit point (actually, the compiler has deemed it would be better to move the exit sequences up the parser tree).

In case the number of cases is big enough but the list of values has "gaps" (i.e. some values are missing and are treated as the `default` case), the jump list will still be formed, but with the address for the `default` case will be used for those entries. There is an internal threshold calculation in the compiler that changes if and how the jump list is generated with respect to the number of cases and the number of sequential values that the compiler can spot in the list of cases.