Computed goto for efficient dispatch tables (https://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables)

July 12, 2012 at 15:44 **Tags** Assembly (https://eli.thegreenplace.net/tag/assembly) , C & C++ (https://eli.thegreenplace.net/tag/c-c)

Recently, while idly browsing through the source code of Python, I came upon an interesting comment in the bytecode VM implementation (`Python/ceval.c`) about using the computed gotos (http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html) extension of GCC [1]. Driven by curiosity, I decided to code a simple example to evaluate the difference between using a computed goto and a traditional `switch` statement for a simple VM. This post is a summary of my findings.

Defining a simple bytecode VM

First let's make clear what I mean by a "VM" in this context - a Bytecode Interpreter (http://en.wikipedia.org/wiki/Interpreter_(computing)). Simply put, it's a loop that chugs through a sequence of instructions, executing them one by one.

Using Python's 2000-line strong (a bunch of supporting macros not included) `PyEval_EvalFrameEx` as an example wouldn't be very educational. Therefore, I'll define a tiny VM whose only state is an integer and has a few instructions for manipulating it. While simplistic, the general structure of this VM is very similar to real-world VMs. This VM is so basic that the best way to explain it is just to show its implementation:

```
#define OP_HALT     0x0
#define OP_INC      0x1
#define OP_DEC      0x2
#define OP_MUL2     0x3
#define OP_DIV2     0x4
#define OP_ADD7     0x5
#define OP_NEG      0x6

int interp_switch(unsigned char* code, int initval) {
    int pc = 0;
    int val = initval;

    while (1) {
        switch (code[pc++]) {
            case OP_HALT:
                return val;
            case OP_INC:
                val++;
                break;
            case OP_DEC:
                val--;
                break;
            case OP_MUL2:
                val *= 2;
                break;
            case OP_DIV2:
                val /= 2;
                break;
            case OP_ADD7:
                val += 7;
                break;
            case OP_NEG:
                val = -val;
                break;
            default:
                return val;
        }
    }
}
```

Note that this is perfectly "standard" C. An endless loop goes through the instruction stream and a `switch` statement chooses what to do based on the instruction opcode. In this example the control is always linear (`pc` only advances by 1 between instructions), but it would not be hard to extend this with flow-control instructions that modify `pc` in less trivial ways.

The `switch` statement should be implemented very efficiently by C compilers - the condition serves as an offset into a lookup table that says where to jump next. However, it turns out that there's a popular GCC extension that allows the compiler to generate even faster code.

Computed gotos

I will cover the details of computed gotos very briefly. For more information, turn to the GCC docs (http://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html) or Google.

Computed gotos is basically a combination of two new features for C. The first is taking addresses of labels into a `void*`.

```
    void* labeladdr = &&somelabel;
somelabel:
    // code
```

The second is invoking `goto` on a variable expression instead of a compile-time-known label, i.e.:

```
void* table[];  // addresses
goto *table[pc];
```

As we'll see shortly, these two features, when combined, can facilitate an interesting alternative implementation of the main VM loop.

To anyone with a bit of experience with assembly language programming, the computed goto immediately makes sense because it just exposes a common instruction that most modern CPU architectures have - jump through a register (aka. indirect jump).

The simple VM implemented with a computed goto

Here's the same VM, this time implemented using a computed goto [2]:

```
int interp_cgoto(unsigned char* code, int initval) {
    /* The indices of labels in the dispatch_table are the relevant opcodes
    */
    static void* dispatch_table[] = {
        &&do_halt, &&do_inc, &&do_dec, &&do_mul2,
        &&do_div2, &&do_add7, &&do_neg};
    #define DISPATCH() goto *dispatch_table[code[pc++]]

    int pc = 0;
    int val = initval;

    DISPATCH();
    while (1) {
        do_halt:
            return val;
        do_inc:
            val++;
            DISPATCH();
        do_dec:
            val--;
            DISPATCH();
        do_mul2:
            val *= 2;
            DISPATCH();
        do_div2:
            val /= 2;
            DISPATCH();
        do_add7:
            val += 7;
            DISPATCH();
        do_neg:
            val = -val;
            DISPATCH();
    }
}
```

Benchmarking

I did some simple benchmarking with random opcodes and the `goto` version is 25% faster than the `switch` version. This, naturally, depends on the data and so the results can differ for real-world programs.

Comments inside the CPython implementation note that using computed goto made the Python VM 15-20% faster, which is also consistent with other numbers I've seen mentioned online.

Why is it faster?

Further down in the post you'll find two "bonus" sections that contain annotated disassembly of the two functions shown above, compiled at the `-O3` optimization level with GCC. It's there for the real low-level buffs among my readers, and as a future reference for myself. Here I aim to explain why the computed goto code is faster at a bit of a higher level, so if you feel there are not enough details, go over the disassembly in the bonus sections.

The computed goto version is faster because of two reasons:

1. The `switch` does a bit more per iteration because of bounds checking.
2. The effects of hardware branch prediction.

Doing less per iteration

If you examine the disassembly of the `switch` version, you'll see that it does the following per opcode:

- Execute the operation itself (i.e. `val *= 2` for `OP_MUL2`)
- `pc++`
- Check the contents of `code[pc]`. If within bounds (`<= 6`), proceed. Otherwise return from the function.
- Jump through the jump table based on offset computed from `code[pc]`.

On the other hand, the computed goto version does this:

- Execute the operation itself
- `pc++`
- Jump through the jump table based on offset computed from `code[pc]`.

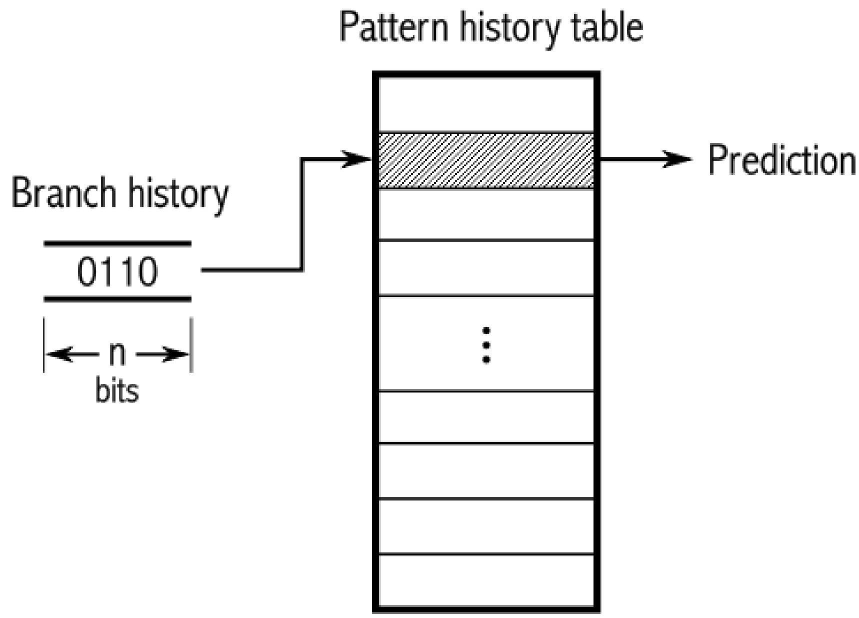
The difference between the two is obviously the "bounds check" step of the `switch`. Why is it required? You may think that this is because of the `default` clause, but that isn't true. Even without the `default` clause, the compiler is forced to generate the bounds check for the `switch` statement to conform to the C standard. Quoting from C99:

> If no converted case constant expression matches and there is no default label, no part of the switch body is executed.

Therefore, the standard forces the compiler to generate "safe" code for the switch. Safety, as usual, has cost, so the `switch` version ends up doing a bit more per loop iteration.

Branch prediction

Modern CPUs have deep instruction pipelines and go to great lengths ensuring that the pipelines stay as full as possible. One thing that can ruin a pipeline's day is a branch, which is why branch predictors (http://en.wikipedia.org/wiki/Branch_predictor) exist. Put simply (read the linked Wikipedia article for more details), it's an algorithm used by the CPU to try to predict in advance whether a branch will be taken or not. Since a CPU can easily pre-fetch instructions from the branch's target, successful prediction can make the pre-fetched instructions valid and there is no need to fully flush the pipeline.



The thing with branch predictors is that they map branches based on their addresses. Since the `switch` statement has a single "master jump" that dispatches all opcodes, predicting its destination is quite difficult. On the other hand, the computed goto statement is compiled into a separate jump per opcode, so given that instructions often come in pairs it's much easier for the branch predictor to "home in" on the various jumps correctly.

Think about it this way: for each jump, the branch predictor keeps a prediction of where it will jump next. If there's a jump per opcode, this is equivalent to predicting the second opcode in an opcode pair, which actually has some chance of success from time to time. On the other hand, if there's just a single jump, the prediction is shared between all opcodes and they keep stepping on each other's toes with each iteration.

I can't say for sure which one of the two factors weighs more in the speed difference between the `switch` and the computed goto, but if I had to guess I'd say it's the branch prediction.

What is done in other VMs?

So this post started by mentioning that the Python implementation uses a computed goto in its bytecode interpreter. What about other VMs?

- Ruby 1.9 (YARV): also uses computed goto.
- Dalvik (the Android Java VM): computed goto
- Lua 5.2: uses a switch
- Finally, if you want to take a look at a simple, yet realistic VM, I invite you to examine the source code of Bobscheme (https://github.com/eliben/bobscheme) - my own Scheme implementation. The "barevm" component (a bytecode interpreter in C++) uses a switch to do the dispatching.

Bonus: detailed disassembly of interp_switch

Here's an annotated disassembly of the `interp_switch` function. The code was compiled with `gcc`, enabling full optimizations (`-O3`).

```
0000000000400650 <interp_switch>:
#
# Per the System V x64 ABI, "code" is in %rdi, "initval" is in %rsi,
# the returned value is in %eax.
#
  400650:     89 f0                   mov    %esi,%eax
#
# This an other NOPx instructions are fillers used for aligning other
# instructions.
#
  400652:     66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
#
# This is the main entry to the loop.
# If code[pc] <= 6, go to the jump table. Otherwise, proceed to return
# from the function.
#
  400658:     80 3f 06                cmpb   $0x6,(%rdi)
  40065b:     76 03                   jbe    400660 <interp_switch+0x10>
#
# Return. This also handles OP_HALT
#
  40065d:     f3 c3                   repz retq
  40065f:     90                      nop
#
# Put code[pc] in %edx and jump through the jump table according to
# its value.
#
  400660:     0f b6 17                movzbl (%rdi),%edx
  400663:     ff 24 d5 20 0b 40 00    jmpq   *0x400b20(,%rdx,8)
  40066a:     66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
#
# Handle OP_ADD7
#
  400670:     83 c0 07                add    $0x7,%eax
  400673:     0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
#
# pc++, and back to check the next opcode.
#
  400678:     48 83 c7 01             add    $0x1,%rdi
  40067c:     eb da                   jmp    400658 <interp_switch+0x8>
  40067e:     66 90                   xchg   %ax,%ax
#
# Handle OP_DIV2
#
  400680:     89 c2                   mov    %eax,%edx
  400682:     c1 ea 1f                shr    $0x1f,%edx
  400685:     8d 04 02                lea    (%rdx,%rax,1),%eax
  400688:     d1 f8                   sar    %eax
  40068a:     eb ec                   jmp    400678 <interp_switch+0x28>
  40068c:     0f 1f 40 00             nopl   0x0(%rax)
#
# Handle OP_MUL2
#
  400690:     01 c0                   add    %eax,%eax
  400692:     eb e4                   jmp    400678 <interp_switch+0x28>
#
# Handle OP_DEC
#
  400694:     0f 1f 40 00             nopl   0x0(%rax)
```

```
  400698:     83 e8 01                sub    $0x1,%eax
  40069b:     eb db                   jmp    400678 <interp_switch+0x28>
  40069d:     0f 1f 00                nopl   (%rax)
#
# Handle OP_INC
#
  4006a0:     83 c0 01                add    $0x1,%eax
  4006a3:     eb d3                   jmp    400678 <interp_switch+0x28>
  4006a5:     0f 1f 00                nopl   (%rax)
#
# Handle OP_NEG
#
  4006a8:     f7 d8                   neg    %eax
  4006aa:     eb cc                   jmp    400678 <interp_switch+0x28>
  4006ac:     0f 1f 40 00             nopl   0x0(%rax)
```

How did I figure out which part of the code handles which opcode? Note that the "table jump" is done with:

```
jmpq   *0x400b20(,%rdx,8)
```

This takes the value in `%rdx`, multiplies it by 8 and uses the result as an offset from `0x400b20`. So the jump table itself is contained at address `0x400b20`, which can be seen by examining the `.rodata` section of the executable:

```
$ readelf -x .rodata interp_compute_gotos

Hex dump of section '.rodata':
  0x00400b00 01000200 00000000 00000000 00000000 ................
  0x00400b10 00000000 00000000 00000000 00000000 ................
  0x00400b20 5d064000 00000000 a0064000 00000000 ].@.......@.....
  0x00400b30 98064000 00000000 90064000 00000000 ..@.......@.....
  0x00400b40 80064000 00000000 70064000 00000000 ..@.....p.@.....
  0x00400b50 a8064000 00000000 01010306 02020405 ..@.............
```

Reading the 8-byte values starting at `0x400b20`, we get the mapping:

```
0x0 (OP_HALT) -> 0x40065d
0x1 (OP_INC)  -> 0x4006a0
0x2 (OP_DEC)  -> 0x400698
0x3 (OP_MUL2) -> 0x400690
0x4 (OP_DIV2) -> 0x400680
0x5 (OP_ADD7) -> 0x400670
0x6 (OP_NEG)  -> 0x4006a8
```

Bonus: detailed disassembly of interp_cgoto

Similarly to the above, here is an annotated disassembly of the `interp_cgoto` function. I'll leave out stuff explained in the earlier snippet, trying to focus only on the things unique to the computed goto implementation.

```
00000000004006b0 <interp_cgoto>:
  4006b0:     0f b6 07                movzbl (%rdi),%eax
#
# Move the jump address indo %rdx from the jump table
#
  4006b3:     48 8b 14 c5 e0 0b 40    mov    0x400be0(,%rax,8),%rdx
  4006ba:     00
  4006bb:     89 f0                   mov    %esi,%eax
#
# Jump through the dispatch table.
#
  4006bd:     ff e2                   jmpq   *%rdx
  4006bf:     90                      nop
#
# Return. This also handles OP_HALT
#
  4006c0:     f3 c3                   repz retq
  4006c2:     66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
#
# Handle OP_INC.
# The pattern here repeats for handling other instructions as well.
# The next opcode is placed into %edx (note that here the compiler
# chose to access the next opcode by indexing code[1] and only later
# doing code++.
# Then the operation is done (here, %eax += 1) and finally a jump
# through the table to the next instruction is performed.
#
  4006c8:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  4006cc:     83 c0 01                add    $0x1,%eax
  4006cf:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  4006d6:     00
  4006d7:     66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
  4006de:     00 00
  4006e0:     48 83 c7 01             add    $0x1,%rdi
  4006e4:     ff e2                   jmpq   *%rdx
  4006e6:     66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
  4006ed:     00 00 00
#
# Handle OP_DEC
#
  4006f0:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  4006f4:     83 e8 01                sub    $0x1,%eax
  4006f7:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  4006fe:     00
  4006ff:     48 83 c7 01             add    $0x1,%rdi
  400703:     ff e2                   jmpq   *%rdx
  400705:     0f 1f 00                nopl   (%rax)
#
# Handle OP_MUL2
#
  400708:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  40070c:     01 c0                   add    %eax,%eax
  40070e:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  400715:     00
  400716:     48 83 c7 01             add    $0x1,%rdi
  40071a:     ff e2                   jmpq   *%rdx
  40071c:     0f 1f 40 00             nopl   0x0(%rax)
#
# Handle OP_DIV2
```

```
#
  400720:     89 c2                   mov    %eax,%edx
  400722:     c1 ea 1f                shr    $0x1f,%edx
  400725:     8d 04 02                lea    (%rdx,%rax,1),%eax
  400728:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  40072c:     d1 f8                   sar    %eax
  40072e:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  400735:     00
  400736:     48 83 c7 01             add    $0x1,%rdi
  40073a:     ff e2                   jmpq   *%rdx
  40073c:     0f 1f 40 00             nopl   0x0(%rax)
#
# Handle OP_ADD7
#
  400740:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  400744:     83 c0 07                add    $0x7,%eax
  400747:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  40074e:     00
  40074f:     48 83 c7 01             add    $0x1,%rdi
  400753:     ff e2                   jmpq   *%rdx
  400755:     0f 1f 00                nopl   (%rax)
#
# Handle OP_NEG
#
  400758:     0f b6 57 01             movzbl 0x1(%rdi),%edx
  40075c:     f7 d8                   neg    %eax
  40075e:     48 8b 14 d5 e0 0b 40    mov    0x400be0(,%rdx,8),%rdx
  400765:     00
  400766:     48 83 c7 01             add    $0x1,%rdi
  40076a:     ff e2                   jmpq   *%rdx
  40076c:     0f 1f 40 00             nopl   0x0(%rax)
```

Again, if we use `readelf` to look at address `0x400be0`, we see the contents of the jump table, and infer the addresses which handle the various opcodes:

```
0x0 (OP_HALT) -> 0x4006c0
0x1 (OP_INC)  -> 0x4006c8
0x2 (OP_DEC)  -> 0x4006f0
0x3 (OP_MUL2) -> 0x400708
0x4 (OP_DIV2) -> 0x400720
0x5 (OP_ADD7) -> 0x400740
0x6 (OP_NEG)  -> 0x400758
```

[1] To the best of my knowledge, it's supported by other major compilers such as ICC and Clang, but not by Visual C++.

[2] Note that the `while` loop here isn't really necessary because the looping is implicitly handled by the `goto` dispatching. I'm leaving it in just for visual consistency with the previous sample.

For comments, please send me an email (mailto:eliben@gmail.com), or reach out on Twitter (https://twitter.com/elibendersky).

Back to top