# The noploop CPU Benchmark

26 Apr 2014

My preference is to write benchmark tools myself, in assembly, and then to dissassemble the compiled machine code for verification. How else do you know what the CPU is actually doing?

It's usually impractical to do this, but I'll share one case where you can. This is for CPU benchmarking, and I use it along with other tools (eg, sysbench, lmbench), and an active benchmarking approach.

It is a procedure for measuring CPU clock speed using an unrolled No-Operation (NOP) loop. It's intended to be simple, minimizing variation caused by cache misses, stall cycles, and branch misprediction. The result provides a baseline before more complex CPU benchmarks are tried.

Since I'm running this on Linux I could just read /proc/cpuinfo, but I don't completely trust it in virtualized environments (which can fake the cpuid). An approach I trust much more is to simply read cycle counters from the CPU Performance Monitoring Unit (eg, using perf), but I have limited or no access to these in virtualized environments. This instance happens to be a Xen guest on AWS EC2.

First, instead of writing assembly from scratch, lets take a shortcut. This C program counts to 10 million:

```
$ vi noploop.c
int
main() {
    int i;
    for (i = 0; i < 10000000; i++) { }
    return (0);
}
```

Now compile to assembly (with -O0, for fewer optimizations), and look for the loop:

```
$ gcc -O0 -S noploop.c
$ cat -n noploop.s
[...]
    12      .cfi_def_cfa_register 6
    13      movl    $0, -4(%rbp)
    14      jmp .L2
    15  .L3:
    16      addl    $1, -4(%rbp)
    17  .L2:
    18      cmpl    $9999999, -4(%rbp)
    19      jle .L3
    20      movl    $0, %eax
    21      popq    %rbp
    22      .cfi_def_cfa 7, 8
[...]
```

The loop became lines 15 to 19. Key lines in English:

- 16 `addl $1, -4(%rpb)`: add of size long the constant 1 to the memory address pointed to by the rpb register minus 4 bytes (this will be the local variable, i).
- 18 `cmpl $999999, -4(%rbp)`: compare of size long the constant 999,999 to the earlier variable.
- 19 `jle .L3`: jump to the .L3 label if the compare was less-than or equal-to.

Now I'll edit the assembly and insert 2,000 NOPs into the loop:

```
$ vi +'set nu' noploop.s
[...]
.L3:
        addl    $1, -4(%rbp)
.L2:
        cmpl    $9999999, -4(%rbp)
        nop
        nop
        nop
[... etc, 2,000 nop's in total ...]
        nop
        jle     .L3
[...]
```

Why 2,000 NOPs? I'm picking a high number so that the CPU spends most of its time running NOP, and cycles spent on tests and branches become negligible. But I'm also not making it too high: these NOPs become 2000 bytes of machine code, which should cache easily and stay within one page of memory. If I made that too high, I might start spending cycles on cache misses and memory page translation.

Compiling to binary and verifying using objdump:

```
$ gcc -o noploop noploop.s
$ objdump -d noploop
[...]
  4004bd:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  4004c1:   81 7d fc 7f 96 98 00    cmpl   $0x98967f,-0x4(%rbp)
  4004c8:   90                      nop
  4004c9:   90                      nop
  4004ca:   90                      nop
  4004cb:   90                      nop
[... etc ...]
  400c97:   90                      nop
  400c98:   0f 8e 1f f8 ff ff       jle    4004bd
[...]
```

Now running:

```
$ time ./noploop

real    0m1.631s
user    0m1.628s
sys     0m0.000s
```

So we ran 20 billion NOPs (10 million x 2000) in 1.628 (user) seconds. For simplicity, I'll treat program startup and loop instructions as negligible, leaving the NOPs. NOPs are executed in a single CPU cycle, so we can calculate the cycle rate: 20 billion cycles / 1.628 seconds = 12.285 billion cycles per second. That would make this a 12.285 GHz processor.

Modern CPUs can execute multiple instructions with each cycle, thanks to the superscalar architecture. How many depends on the instruction and CPU functional units that can operate in parallel; for NOPs it's likely to be 3 or 4 (with newer processors). In this case, it's 4, making my actual CPU clock speed 12.285 / 4 = 3.071 GHz.

This may be higher than expected due to Intel Turbo Boost (or any similar equivalent), a processor feature that can dynamically overclock the CPUs depending on factors including temperature. For consistent benchmarking, you can turn it off in the BIOS, bearing in mind that the CPUs will now run more slowly. Linux may also mess with CPU frequencies, using a different interface (eg, Intel SpeedStep), run by cpufreqd. Remember to either turn them all off or account for the variation.

This is just one CPU benchmark, aimed at providing supporting evidence for further study. I also like to run the mhz tool from lmbench:

```
$ mhz
3097 MHz, 0.3229 nanosec clock
```

Which uses a similar approach: running some instructions with expected cycle counts, and timing them.

So what did /proc/cpuinfo claim?

```
$ grep model.name /proc/cpuinfo
model name  : Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
[...]
```

This could well be accurate, as 3 GHz is within the Turbo Boost envelope for the E5-2680v2.

---

*You can comment here, but I can't guarantee your comment will remain here forever: I might switch comment systems at some point (eg, if disqus add advertisements).*