Popcount and the Black Art Of Microbenchmarking

Bart Massey
Portland C/Assembly Users Group
2014-03-11
bart.massey@gmail.com

In 1984, when I learned C...

- Everyone was concerned about performance. After all, this was the HW of the day:

Because performance

- It was *de rigeur* to optimize your C code.
  - This was especially important since with PCC you were basically writing assembly.
  - Besides, programs were tiny, and CPUs were *tres* predictable, so it was easy.

One might want to know…

- For example, the number of 1-bits in a machine word. (For VAX, a 32-bit word!)

  12345678 = 0xbc614e =
  0b00000000101111000110000101001110

  has 12 1-bits.

  popcount(12345678) == 12

There's the obvious slow way

```
uint32 popcount(uint32 n) {
    int p = 0;
    for (int i = 0; i < 32; i++)
        p += (n >> i) & 1;
    return p;
}
```

Slow way is slow

- Must be optimization opportunities?
- (Why we care about popcount:
  - e.g. graphics: density of bitmap
  - e.g. machine learning: mismatched bits
  - e.g. adversary search: mobility
  - e.g. crypto: IDK)
- Why is there not a popcount instruction?

What does speed mean anyway?

- But how do we measure speed?
- Obvious answer: microbenchmark

```
int main( ) {
    for (int i = 0; i < 100000000; i++)
        popcount(0);
    return 0;
}
```

4 easy microbenchmark fails

- Not test over enough domain.
- Lose control of the optimizer.
- Lose precision in measurement.
- Benchmark non-working code.

3 hard fails + microbenchmarks are evil

- Not set up a “realistic” environment for microbenchmark code.
- Make code very environment-sensitive.
- Elevate importance of small differences.

- Net result: near-meaningless comparisons. Most (all) microbenchmarks are here.

My Story

- XCB needed popcount. Could have used X11 popcount, which was HAKMEM 169. But...ugh.
- Goals: Reasonably fast. In worst case. Really portable performance (run anywhere fast). Simple enough to understand / maintain.
- Off I go: survey and microbenchmark.

http://github.com/BartMassey/popcount

Key idea: bit parallelism

- Your CPU is a parallel computer (64-way)
- The bible of this stuff:

  Hacker's Delight
  Henry S. Warren Jr.

- Example: Do X and Y differ in exactly one bit-position?

```
a = X ^ Y
a != 0 && (a & (a - 1)) == 0
```

Key idea: doubling

- Imagine we had four eight-bit popcounts at positions 0, 8, 16, 24 in our 32-bit word.
  - popcounts range from 0-8, so at most 4 bits.
    P1 P2 P3 P4
  - Can add a pair of them to be at most 5 bits.
    p = p + (p >> 8)
    G (P1+P2) G (P3+P4)
  - Can add a pair again to be at most 6 bits.
    p = p + (p >> 16)
    G G G (P1+P2+P3+P4)
  - Now just trash the garbage.
    p = p & 0xff

Tricks for doubling

- Double all the way using masks.
- Lots of slight variants on this scheme.
- Can do the final combination (previous slide) in one integer multiply, if that's faster.

HAKMEM 169

- “Casting out sixty-threes” approach.
- Still some binary steps.
- Several variants: all hairy.
- Slow anyway.

Table-driven popcount

- Basic approach: precompute all popcounts of 8 (16) bits. Then do 2 (1) doubling steps to combine them.
- Fastest in microbenchmarks.
- Bad problems in non-microbenchmarks:
  - Victimized by cache pressure.
  - Causes cache pressure.
  - Cost of precompute or readin.

Domain & Optimization

- Important to run over a variety of inputs.
- Important to ensure that the program prints an answer that is a function of all inputs.
- Important to ensure that optimizations chosen by the compiler are “fair”.

Driver structure: macro-gen

- Code for the actual driver loop is generated as a macro.
- Actual calls to popcount() are inlined.
- Thus, bulk of time is spent in popcount().

Timing

- Wall-clock time is almost always best.
- Be careful with computation from horrible UNIX struct timevals.
- Report in consistent units: e.g. nanoseconds
- Watch out for inter-run variance
  - Not a problem here.

Final Thoughts

- If you do everything just so and measure carefully…it doesn’t matter what you do as long as it’s not stupid.
- This is a general rule.
- Take-Home lessons:
  - Microbenchmarks are evil and easy to screw up.
  - Bit parallelism is cool, and C is great for it.