Daniel Lemire is a computer science professor at the University of Quebec (Canada). His research is focused on software performance and data engineering. He is a techno-optimist.

Faster remainders when the divisor is a constant: beating compilers and libdivide

Not all instructions on modern processors cost the same. Additions and subtractions are cheaper than multiplications which are themselves cheaper than divisions. For this reason, compilers frequently replace division instructions by multiplications. Roughly speaking, it works in this manner. Suppose that you want to divide a variable `n` by a constant `d`. You have that `n/d = n * (2^N/d) / (2^N)`. The division by a power of two (`/ (2^N)`) can be implemented as a right shift if we are working with unsigned integers, which compiles to single instruction: that is possible because the underlying hardware uses a base 2. Thus if $2^N/d$ has been precomputed, you can compute the division `n/d` as a multiplication and a shift. Of course, if `d` is not a power of two, $2^N/d$ cannot be represented as an integer. Yet for `N` large enough[footnote], we can approximate $2^N/d$ by an integer and have the exact computation of the remainder for all possible `n` within a range. I believe that all optimizing C/C++ compilers know how to pull this trick and it is generally beneficial irrespective of the processor's architecture.

The idea is not novel and goes back to at least 1973 (Jacobsohn). However, engineering matters because computer registers have finite number of bits, and multiplications can overflow. I believe that, historically, this was first introduced into a major compiler (the GNU GCC compiler) by Granlund and Montgomery (1994). While GNU GCC and the Go compiler still rely on the approach developed by Granlund and Montgomery, other compilers like LLVM's clang use a slightly improved version described by Warren in his book Hacker's Delight.

What if `d` is a constant, but not known to the compiler? Then you can use a library like libdivide. In some instances, libdivide can even be more efficient than compilers because it uses an approach introduced by Robison (2005) where we not only use multiplications and shifts, but also an addition to avoid arithmetic overflows.

Can we do better? It turns out that in some instances, we can beat both the compilers and a library like libdivide.

Everything I have described so far has to do with the computation of the quotient (`n/d`) but quite often, we are looking for the remainder (noted `n % d`). How do compilers compute the remainder? They first compute the quotient `n/d` and then they multiply it by the divisor, and subtract all of that from the original value (using the identity `n % d = n - (n/d) * d`).

Can we take a more direct route? We can.

Let us go back to the intuitive formula `n/d = n *` $(2^N/d)$ `/` (2^N). Notice how we compute the multiplication and then drop the least significant `N` bits? It turns out that if, instead, we keep these least significant bits, and multiply them by the divisor, we get the remainder, directly without first computing the quotient.

The intuition is as follows. To divide by four, you might choose to multiply by 0.25 instead. Take 5 * 0.25, you get 1.25. The integer part (1) gives you the quotient, and the decimal part (0.25) is indicative of the remainder: multiply 0.25 by 4 and you get 1, which is the remainder. Not only is this more direct and potential useful in itself, it also gives us a way to check quickly whether the remainder is zero. That is, it gives us a way to check that we have an integer that is divisible by another: do `x * 0.25`, the decimal part is less than 0.25 if and only if `x` is a multiple of 4.

This approach was known to Jacobsohn in 1973, but as far as I can tell, he did not derive the mathematics. Vowels in 1994 worked it out for the case where the divisor is 10, but (to my knowledge), nobody worked out the general case. It has now been worked out in a paper to appear in Software: Practice and Experience called Faster Remainder by Direct Computation.

In concrete terms, here is the C code to compute the remainder of the division by some fixed divisor `d`:

```
uint32_t d = ...;// your divisor > 0

uint64_t c = UINT64_C(0xFFFFFFFFFFFFFFFF) / d + 1;

// fastmod computes (n mod d) given precomputed c
uint32_t fastmod(uint32_t n ) {
  uint64_t lowbits = c * n;
  return ((__uint128_t)lowbits * d) >> 64;
```

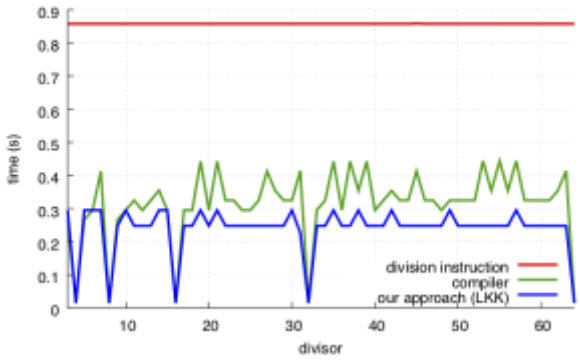
```
}
```

The divisibility test is similar...

```
uint64_t c = 1 + UINT64_C(0xffffffffffffffff) / d;


// given precomputed c, checks whether n % d == 0
bool is_divisible(uint32_t n) {
  return n * c <= c - 1;
}
```

To test it out, we did many things, but in one particular tests, we used a hashing function that depends on the computation of the remainder. We vary the divisor and compute many random values. In one instance, we make sure that the compiler cannot assume that the divisor is known (so that the division instruction is used), in another case we let the compiler do its work, and finally we plug in our function. On a recent Intel processor (Skylake), we beat state-of-the-art compilers (e.g., LLVM's clang, GNU GCC).



The computation of the remainder is nice, but I really like better the divisibility test. Compilers generally don't optimize divisibility tests very well. A line of code like `(n % d) = 0` is typically compiled to the computation of the remainder (`(n % d)`) and a test to see whether it is zero. Granlund and Montgomery have a better approach if `d` is known ahead of time and it involves computing the inverse of an odd integer using Newton's method. Our approach is simpler and faster (on all tested platforms) in our tests. It is a multiplication by a constant followed by a comparison of the result with said constant: it does not get much cheaper than that. It seems that compilers could easily apply such an approach.

We packaged the functions as part of a header-only library which works with all major C/C++ compilers (GNU GCC, LLVM's clang, Visual Studio). We also published our benchmarks for research purposes.

I feel that the paper is short and to the point. There is some mathematics, but we worked hard so that it is as easy to understand as possible. And don't skip the introduction! It tells a nice story.

The paper contains carefully crafted benchmarks, but I came up with a fun one for this blog post which I call "fizzbuzz". Let us go through all integers in sequence and count how many are divisible by 3 and how many are divisible by 5. There are far more efficient ways to do that, but here is the programming 101 approach in C:

```
  for (uint32_t i = 0; i < N; i++) {
    if ((i % 3) == 0)
      count3 += 1;
    if ((i % 5) == 0)
      count5 += 1;
  }
```

Here is the version with our approach:

```
static inline bool is_divisible(uint32_t n, uint64_t M) {
  return n * M <= M - 1;
}

...


  uint64_t M3 = UINT64_C(0xFFFFFFFFFFFFFFFF) / 3 + 1;
  uint64_t M5 = UINT64_C(0xFFFFFFFFFFFFFFFF) / 5 + 1;
  for (uint32_t i = 0; i < N; i++) {
    if (is_divisible(i, M3))
      count3 += 1;
    if (is_divisible(i, M5))
      count5 += 1;
  }
```

Here is the number of CPU cycles spent on each integer checked (average):

Compiler	4.5 cycles per integer
Fast approach	1.9 cycles per integer

I make my benchmarking code available. For this test, I am using an Intel (skylake) processing and GCC 8.1.

Your results will vary. Our proposed approach may not always be faster. However, we can claim that some of time, it is advantageous.

Further reading: Faster Remainder by Direct Computation: Applications to Compilers and Software Libraries, Software: Practice and Experience (to appear)

Footnote: What is `N`? If both the numerator `n` and the divisor `d` are 32-bit unsigned integers, then you can pick `N=64`. This is not the smallest possible value. The smallest possible value is given by Algorithm 2 in our paper and it involves a bit of mathematics (note: the notation in my blog post differs from the paper, `N` becomes `F`).

PUBLISHED BY

Daniel Lemire
A computer science professor at the Université du Québec (TELUQ).
View all posts by Daniel Lemire →

C++, division, modulo, performance

20 thoughts on “Faster remainders when the divisor is a constant: beating compilers and libdivide”

James
February 8, 2019 at 6:57 pm

Your first paragraph states “multiplications which are themselves slower than divisions”, but surely you meant the opposite.

Daniel Lemire
February 8, 2019 at 7:29 pm

Yes. Thank you, I fixed that.

Denis Bakhvalov
February 8, 2019 at 7:19 pm

Looks like it's worth to try to implement in the compilers.

Brian m
February 9, 2019 at 5:20 am

I ageee. This sounds like a great patch which would be integrated into llvm and/or gcc, rather than a library that would need to be included in. Its great to test to the approach, but realistically most people writing code will not use your library only because they dont know it exists…. It would be better if just a normal part of the compile process (in the compiler itself)

Marco
February 9, 2019 at 12:10 am

Can javascript do the same function? Thanks

Daniel Lemire ▲
February 9, 2019 at 12:56 am

This is applicable to JavaScript. Integer arithmetic in JavaScript is tricky so you would need to focus on a specific problem.

Evidently, this trick could be implemented in the JavaScript compiler.

Marco
February 9, 2019 at 1:13 am

Hi Daniel, you mean pure javascript code can do that?
Can you explain more , Thanks

Marco
February 9, 2019 at 1:44 am

Hi Daniel,i have try below javascript code,but get the wrong result ,can you explain why,Thanks.

function fmod(divisor,num){
let c=(0xFFFFFFFFFFFFFFFF)/divisor+1;
let lowbits=c*num;*
*return ((lowbits*divisor)>>64)
}

console.log(fmod(5/18))

Damian Gray

February 9, 2019 at 2:09 am

All numbers in JS are floats under the hood so the raw bits are not going to act the same as ints. That's probably why the function isn't working as is in pure JS.

Sol

February 9, 2019 at 2:36 am

Numbers in JavaScript are stored as doubles, and thus cannot exactly represent an integer over 53 bits. Additionally, the bitwise operators all convert the operands to 32-bit signed integers before performing the operation. You are doubly unable to do 64-bit shifts in JavaScript.

Nathan

February 9, 2019 at 3:28 am

The most likely answer is that JavaScript does not use 64 bit integer numbers. It uses 64 bit floating point numbers which happen to look like integers for most development purposes.

Severin Pappadeux

February 9, 2019 at 1:40 am

Looks like https://math.stackexchange.com/questions/1251327/is-there-a-fast-

divisibility-check-for-a-fixed-divisor

Daniel Lemire

February 9, 2019 at 2:30 pm

It is a similar idea regarding the divisibility test but it does not look the same to me. Our divisibility test has a specific form (`n * M < M`). Also the answer seems to differentiate between odd and even integers, we do not. Please see our manuscript for the mathematical derivations… https://arxiv.org/abs/1902.01961

Steve

February 9, 2019 at 2:01 am

Great article and paper! Small typo: “This approach was known Jacobsohn in 1973” should be “This approach was known TO Jacobsohn in 1973”.

Daniel Lemire

February 9, 2019 at 12:40 pm

I fixed the typo, thanks.

Alexander Monakov

February 9, 2019 at 11:00 am

Coincidentally, in September 2018 a slightly extended version of divisibility check shown in Hacker’s Delight was implemented in GCC (so it doesn’t appear in gcc-8, but will be available in gcc-9). Some discussion can be found in GCC PR 82853, and you can see the optimization in action on Compiler Explorer.

degski

February 9, 2019 at 12:59 pm

All well and good, iff one doesn’t need the quotient [unless I missed something].

Theo

February 9, 2019 at 2:36 pm

Please, have you also tried it with 64-bits “n”, as it would need 192-bits intermediate computation (if I understood correctly)? Can yo perhaps share some timings as well, in addition to the 32-bits case?

PS: Intel’s Cannon Lake is supposed to have 10-18 cycles latency for integer division, could make for interesting comparison!

Daniel Lemire

February 9, 2019 at 3:15 pm

Please, have you also tried it with 64-bits "n", as it would need 192-bits intermediate computation (if I understood correctly)? Can yo perhaps share some timings as well, in addition to the 32-bits case?

You do not need 192-bit intermediate computations, 128-bit is enough. However, the engineering gets more complicated if you are to deal with overflows properly. We are inviting people to pursue the research. We do not claim to have written the last word on this. If you are interested in pursuing the research and want to chat, do get in touch with us.

Intel's Cannon Lake is supposed to have 10-18 cycles latency for integer division, could make for interesting comparison!

I have a CannonLake processor right here, and here are the results of the fizzbuzz benchmark:

```
$ ./fizzbuzz
count35(N, &count3, &count5)                :  3.690 cycles per input
word (best)  3.698 cycles per input word (avg)
1666666700 1000000000
fastcount35(N, &count3, &count5)            :  2.082 cycles per input
word (best)  2.085 cycles per input word (avg)
1666666700 1000000000
```

(This is GCC 8.1.)

Of course, these fizzbuzz tests do not use the division instruction. The division instruction would be slower. However, it would be less terrible than on skylake processors.

Theo

February 9, 2019 at 3:21 pm

Thank you for the reply!