Tricks to speed-up universal hashing

The most popular almost-universal hash function requires the evaluation of a polynomial over an integer field. Two key techniques to making this efficient is divisionless modular reduction and Horner's method for evaluation.

Divisionless mod

Division is the most expensive integer operation on a computer. On many CPUs, addition can take a single clock cycle to perform while division takes dozens. When performing high-speed hashing, division is to be avoided.

Let's say you need to compute x mod y. You can produce a value that is congruent to x mod y as follows.

1. Express y as 2^a-b for some a and b. The technique is more efficient the smaller b is, so it's helpful if y is just a bit smaller than a power of 2. The "Mersenne" primes, which are all of the form 2^a-1, are especially useful in cryptography.
2. Rewrite x as $(x_{hi})(2^a)+(x_{lo})$. This can always be done by letting x_{lo} = x mod 2^a and letting x_{hi} = x div 2^a ("div" here is integer division, ie, the whole number of times 2^a goes into x). On a computer, x_{lo} is simply the low a bits of x and x_{hi} is the rest of the bits.
3. At this point x mod y = $[(x_{hi})(2^a)+(x_{lo})]$ mod $(2^a$-b) for the a, b, x_{lo}, and x_{hi} that you've identified. Note that you can apply mod to intermediate terms at will to simplify terms, which means 2^a can be replaced with 2^a mod $(2^a$-b) in this context, which is just b (2^a-b goes into 2^a once and leaves a remainder of b).

The end result is that x mod y = $[(x_{hi})(b)+(x_{lo})]$ mod $(2^a$-b) where x_{lo} is the low a bits of x and x_{hi} is the rest of the bits. The end result of this process is a number that if modded by y would give the correct result of x mod y. It is thus interchangeable with the true x mod y when used as an intermediate value. When a final result is required, however, a true mod y would need to be computed.

Example: 10,000 mod 255. 10,000 in binary is 10 0111 0001 0000. 255 is 2^8-1 (ie, a=8 and b=1). x_{lo} = 0001 0000, the low 8 bits, and x_{hi} = 10 0111, the rest of the bits. This means 10,000 mod 255 = $[(x_{hi})(b)+(x_{lo})]$ mod 255 = [(39)(1)+16] mod 255 = 55. Because 55 < 255, this is the actual result of 10,000 mod 255 and not just a congruent value.

Example: 500 mod 30. 500 in binary is 1 1111 0100. 30 is 2^5-2 (ie, a=5 and b=2). x_{lo} = 1 0100, the low 5 bits, and x_{hi} = 1111, the rest of the bits. This means 500 mod 30 = $[(x_{hi})(b)+(x_{lo})]$ mod 30 = [(15)(2)+20] mod 30 = 50 mod 30. Because 50 ≥ 30, this is not the actual result of 500 mod 30 but just a congruent value. To get the value to a bits, we can continue this process until the result is 5 bits or less.

50 in binary is 11 0010. 30 is 2^5-2 (ie, a=5 and b=2). x_{lo} = 1 0010, the low 5 bits, and x_{hi} = 1, the rest of the bits. This means 50 mod 30 = $[(x_{hi})(b)+(x_{lo})]$ mod 30 = [(1)(2)+18] mod 30 = 20 mod 30. Because 20 < 30, this is the actual result of 50 mod 30 (and thus 500 mod 30) and not just a

congruent value.

This can be turned into pseudocode:

```
divisionless(x,a,b):        // return value congruet to x mod 2^a-b
   xlo = x & ((1 << a)-1)   // use mask to grab low a bits
   xhi = x >> a             // xhi is bits beyond the low a
   return xhi * b + xlo
```

Horner's method

Consider the polynomial $a_1k^n + a_2k^{n-1} + a_3k^{n-2} + \ldots\ a_nk^1$. A naive implementation might call pow(k,x) for each x = 1 ... n. This would be wasteful. Horner suggested the following algorithm instead.

```
horner(a[1..n], k):   // returns evaluation of polynomial given above
   acc = 0
   for i = 1 to n:
      acc += a[i]
      acc *= k
   return acc
```

How does it work? The first iteration computes $(0 + a_1k) = a_1k$. The second iteration then does one add and one multiply to get $(a_1k + a_2)k = a_1k^2 + a_2k$. The third iteration then does one add and one multiply to get $(a_1k^2 + a_2k + a_3)k = a_1k^3 + a_2k^2 + a_3k$. In general, the i-th iteration increases the degree of each term from the prior iteration's polynomial and adds another term a_ik^{n+1-i}. After the n-th iteration, the desired polynomial has been computed.

The benefits of this process are twofold. The polynomial is evaluated in just n additions and n multiplications, which is minimal, but also the length n is not needed at the begining of the computation. The loop could be written `while in.hasNext()` which would continue the loop as long as there are more coefficients to process. This is an important property for streaming data, where you don't know how long your data is until it terminates.