Abstract

I offer you a new hash function for hash table lookup that is faster and more thorough than the one you are using now. I also give you a way to verify that it is more thorough.

All the text in this color wasn't in the 1997 Dr Dobbs article. The code given here are all public domain.

The Hash

Over the past two years I've built a general hash function for hash table lookup. Most of the two dozen old hashes I've replaced have had owners who wouldn't accept a new hash unless it was a plug-in replacement for their old hash, and was demonstrably better than the old hash.

These old hashes defined my requirements:

- The keys are unaligned variable-length byte arrays.
- Sometimes keys are several such arrays.
- Sometimes a set of independent hash functions were required.
- Average key lengths ranged from 8 bytes to 200 bytes.
- Keys might be character strings, numbers, bit-arrays, or weirder things.
- Table sizes could be anything, including powers of 2.
- The hash must be faster than the old one.
- The hash must do a good job.

Without further ado, here's the fastest hash I've been able to design that meets all the requirements. The comments describe how to use it.

Update: I'm leaving the old hash in the text below, but it's obsolete, I have faster and more thorough hashes now. http://burtleburtle.net/bob/c/lookup3.c (2006) is about 2 cycles/byte, works well on 32-bit platforms, and can produce a 32 or 64 bit hash. SpookyHash (2011) is specific to 64-bit platforms, is about 1/3 cycle per byte, and produces a 32, 64, or 128 bit hash.

```
typedef  unsigned long  int  ub4;   /* unsigned 4-byte quantities */
typedef  unsigned       char ub1;   /* unsigned 1-byte quantities */

#define hashsize(n) ((ub4)1<<(n))
#define hashmask(n) (hashsize(n)-1)

/*
--------------------------------------------------------------------
mix -- mix 3 32-bit values reversibly.
For every delta with one or two bits set, and the deltas of all three
  high bits or all three low bits, whether the original value of a,b,c
  is almost all zero or is uniformly distributed,
* If mix() is run forward or backward, at least 32 bits in a,b,c
  have at least 1/4 probability of changing.
* If mix() is run forward, every bit of c will change between 1/3 and
  2/3 of the time.  (Well, 22/100 and 78/100 for some 2-bit deltas.)
mix() was built out of 36 single-cycle latency instructions in a
  structure that could supported 2x parallelism, like so:
      a -= b;
      a -= c; x = (c>>13);
      b -= c; a ^= x;
      b -= a; x = (a<<8);
      c -= a; b ^= x;
      c -= b; x = (b>>13);
      ...
  Unfortunately, superscalar Pentiums and Sparcs can't take advantage
  of that parallelism.  They've also turned some of those single-cycle
  latency instructions into multi-cycle latency instructions.  Still,
  this is the fastest good hash I could find.  There were about 2^^68
  to choose from.  I only looked at a billion or so.
```

```
--------------------------------------------------------------------
*/
#define mix(a,b,c) \
{ \
  a -= b; a -= c; a ^= (c>>13); \
  b -= c; b -= a; b ^= (a<<8); \
  c -= a; c -= b; c ^= (b>>13); \
  a -= b; a -= c; a ^= (c>>12);  \
  b -= c; b -= a; b ^= (a<<16); \
  c -= a; c -= b; c ^= (b>>5); \
  a -= b; a -= c; a ^= (c>>3);  \
  b -= c; b -= a; b ^= (a<<10); \
  c -= a; c -= b; c ^= (b>>15); \
}

/*
--------------------------------------------------------------------
hash() -- hash a variable-length key into a 32-bit value
  k       : the key (the unaligned variable-length array of bytes)
  len     : the length of the key, counting by bytes
  initval : can be any 4-byte value
Returns a 32-bit value.  Every bit of the key affects every bit of
the return value.  Every 1-bit and 2-bit delta achieves avalanche.
About 6*len+35 instructions.

The best hash table sizes are powers of 2.  There is no need to do
mod a prime (mod is sooo slow!).  If you need less than 32 bits,
use a bitmask.  For example, if you need only 10 bits, do
  h = (h & hashmask(10));
In which case, the hash table should have hashsize(10) elements.

If you are hashing n strings (ub1 **)k, do it like this:
  for (i=0, h=0; i<n; ++i) h = hash( k[i], len[i], h);

By Bob Jenkins, 1996.  bob_jenkins@burtleburtle.net.  You may use this
code any way you wish, private, educational, or commercial.  It's free.

See http://burtleburtle.net/bob/hash/evahash.html
Use for hash table lookup, or anything where one collision in 2^^32 is
acceptable.  Do NOT use for cryptographic purposes.
--------------------------------------------------------------------
*/

ub4 hash( k, length, initval)
register ub1 *k;        /* the key */
register ub4  length;   /* the length of the key */
register ub4  initval;  /* the previous hash, or an arbitrary value */
{
   register ub4 a,b,c,len;

   /* Set up the internal state */
   len = length;
   a = b = 0x9e3779b9;  /* the golden ratio; an arbitrary value */
   c = initval;         /* the previous hash value */

   /*---------------------------------------- handle most of the key */
   while (len >= 12)
   {
      a += (k[0] +((ub4)k[1]<<8) +((ub4)k[2]<<16) +((ub4)k[3]<<24));
      b += (k[4] +((ub4)k[5]<<8) +((ub4)k[6]<<16) +((ub4)k[7]<<24));
      c += (k[8] +((ub4)k[9]<<8) +((ub4)k[10]<<16)+((ub4)k[11]<<24));
      mix(a,b,c);
      k += 12; len -= 12;
   }

   /*------------------------------------- handle the last 11 bytes */
   c += length;
   switch(len)              /* all the case statements fall through */
   {
```

```
   case 11: c+=((ub4)k[10]<<24);
   case 10: c+=((ub4)k[9]<<16);
   case 9 : c+=((ub4)k[8]<<8);
      /* the first byte of c is reserved for the length */
   case 8 : b+=((ub4)k[7]<<24);
   case 7 : b+=((ub4)k[6]<<16);
   case 6 : b+=((ub4)k[5]<<8);
   case 5 : b+=k[4];
   case 4 : a+=((ub4)k[3]<<24);
   case 3 : a+=((ub4)k[2]<<16);
   case 2 : a+=((ub4)k[1]<<8);
   case 1 : a+=k[0];
     /* case 0: nothing left to add */
   }
   mix(a,b,c);
   /*-------------------------------------------- report the result */
   return c;
}
```

Most hashes can be modeled like this:

```
  initialize(internal state)
  for (each text block)
  {
    combine(internal state, text block);
    mix(internal state);
  }
  return postprocess(internal state);
```

In the new hash, mix() takes 3n of the 6n+35 instructions needed to hash n bytes. Blocks of text are combined with the internal state (a,b,c) by addition. This combining step is the rest of the hash function, consuming the remaining 3n instructions. The only postprocessing is to choose c out of (a,b,c) to be the result.

Three tricks promote speed:

1. Mixing is done on three 4-byte registers rather than on a 1-byte quantity.
2. Combining is done on 12-byte blocks, reducing the loop overhead.
3. The final switch statement combines a variable-length block with the registers a,b,c without a loop.

The golden ratio really is an arbitrary value. Its purpose is to avoid mapping all zeros to all zeros.

The Hash Must Do a Good Job

The most interesting requirement was that the hash must be better than its competition. What does it mean for a hash to be good for hash table lookup?

A good hash function distributes hash values uniformly. If you don't know the keys before choosing the function, the best you can do is map an equal number of possible keys to each hash value. If keys were distributed uniformly, an excellent hash would be to choose the first few bytes of the key and use that as the hash value. Unfortunately, real keys aren't uniformly distributed. Choosing the first few bytes works quite poorly in practice.

The real requirement then is that a good hash function should distribute hash values uniformly for the keys that users actually use.

How do we test that? Let's look at some typical user data. (Since I work at Oracle, I'll use Oracle's standard example: the EMP table.) The EMP table. Is this data uniformly distributed?

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30

7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7898	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7539	01-MAY-81	2850		30
7782	CLARK	MANAGER	7566	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7698	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500		30
7876	ADAMS	CLERK	7788	23-MAY-87	1100	0	20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Consider each horizontal row to be a key. Some patterns appear.

1. Keys often differ in only a few bits. For example, all the keys are ASCII, so the high bit of every byte is zero.
2. Keys often consist of substrings arranged in different orders. For example, the MGR of some keys is the EMPNO of others.
3. Length matters. The only difference between zero and no value at all may be the length of the value. Also, "aa aaa" and "aaa aa" should hash to different values.
4. Some keys are mostly zero, with only a few bits set. (That pattern doesn't appear in this example, but it's a common pattern.)

Some patterns are easy to handle. If the length is included in the data being hashed, then lengths are not a problem. If the hash does not treat text blocks commutatively, then substrings are not a problem. Strings that are mostly zeros can be tested by listing all strings with only one bit set and checking if that set of strings produces too many collisions.

The remaining pattern is that keys often differ in only a few bits. If a hash allows small sets of input bits to cancel each other out, and the user keys differ in only those bits, then all keys will map to the same handful of hash values.

A common weakness

Usually, when a small set of input bits cancel each other out, it is because those input bits affect only a smaller set of bits in the internal state.

Consider this hash function:

```
  for (hash=0, i=0; i<hash; ++i)
    hash = ((hash<<5)^(hash>>27))^key[i];
  return (hash % prime);
```

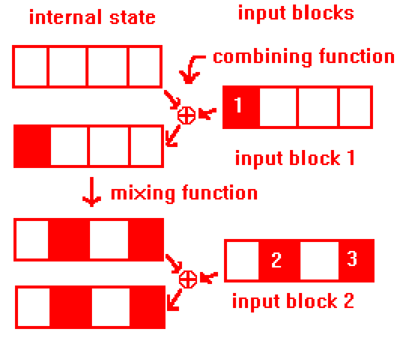
This function maps the strings "EXXXXXB" and "AXXXXXC" to the same value. These keys differ in bit 3 of the first byte and bit 1 of the seventh byte. After the seventh bit is combined, any further postprocessing will do no good because the internal states are already the same.

Any time n input bits can only affect m output bits, and n > m, then the 2^n keys that differ in those input bits can only produce 2^m distinct hash values. The same is true if n input bits can only affect m bits of the internal state -- later mixing may make the 2^m results look uniformly distributed, but there will still be only 2^m results.

The function above has many sets of 2 bits that affect only 1 bit of the internal state. If there are n input bits, there are (n choose 2)=(n*n/2 - n/2) pairs of input bits, only a few of which match weaknesses in the function above. It is a common pattern for keys to differ in only a few bits. If those bits match one of a hash's weaknesses, which is a rare but not negligible event, the hash will do extremely bad. In most cases, though, it will do just fine. (This allows a function to slip through sanity checks, like hashing an English dictionary uniformly, while still frequently bombing on user data.)

In hashes built of repeated combine-mix steps, this is what usually causes this weakness:

1. A small number of bits y of one input block are combined, affecting only y bits of the internal state. So far so good.
2. The mixing step causes those y bits of the internal state to affect only z bits of the internal state.
3. The next combining step overwrites those bits with z more input bits, cancelling out the first y input bits.



A funnel of 3 input bits into 2 output bits

When z is smaller than the number of bits in the output, then y+z input bits have affected only z bits of the internal state, causing 2^{y+z} possible keys to produce at most 2^z hash values.

The same thing can happen in reverse:

1. Uncombine this block, causing y block bits to unaffect y bits of the internal state.
2. Unmix the internal state, leaving x bits unaffected by the y bits from this block.
3. Unmixing the previous block unaffects those x bits, cancelling out this block's y bits.

If x is less than the number of bits in the output, then the 2^{x+y} keys differing in only those x+y input bits can produce at most 2^x hash values.

(If the mixing function is not a permutation of the internal state, it is not reversible. Instead, it loses information about the earlier blocks every time it is applied, so keys differing only in the first few input blocks are more likely to collide. The mixing function ought to be a permutation.)

It is easy to test whether this weakness exists: if the mixing step causes any bit of the internal state to affect fewer bits of the internal state than there are output bits, the weakness exists. This test should be run on the reverse of the mixing function as well. It can also be run with all sets of 2 internal state bits, or all sets of 3.

Another way this weakness can happen is if any bit in the final input block does not affect every bit of the output. (The user might choose to use only the unaffected output bit, then that's 1 input bit that affects 0 output bits.)

A Survey of Hash Functions

We now have a new hash function and some theory for evaluating hash functions. Let's see how various hash functions stack up.

Additive Hash

```
ub4 additive(char *key, ub4 len, ub4 prime)
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash += key[i];
  return (hash % prime);
}
```

This takes 5*n*+3 instructions. There is no mixing step. The combining step handles one byte at a time. Input bytes commute. The table length must be prime, and can't be much bigger than one byte because the value of variable `hash` is never much bigger than one byte.

Rotating Hash

```
ub4 rotating(char *key, ub4 len, ub4 prime)
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash<<4)^(hash>>28)^key[i];
  return (hash % prime);
}
```

This takes 8*n*+3 instructions. This is the same as the additive hash, except it has a mixing step (a circular shift by 4) and the combining step is exclusive-or instead of addition. The table size is a prime, but the prime can be any size. On machines with a rotate (such as the Intel x86 line) this is 6*n*+2 instructions. I have seen the `(hash % prime)` replaced with

```
  hash = (hash ^ (hash>>10) ^ (hash>>20)) & mask;
```

eliminating the % and allowing the table size to be a power of 2, making this 6*n*+6 instructions. % can be very slow, for example it is 230 times slower than addition on a Sparc 20.

One-at-a-Time Hash

```
ub4 one_at_a_time(char *key, ub4 len)
{
  ub4   hash, i;
  for (hash=0, i=0; i<len; ++i)
  {
    hash += key[i];
    hash += (hash << 10);
    hash ^= (hash >> 6);
  }
  hash += (hash << 3);
  hash ^= (hash >> 11);
  hash += (hash << 15);
  return (hash & mask);
}
```

This is similar to the rotating hash, but it actually mixes the internal state. It takes 9*n*+9 instructions and produces a full 4-byte result. Preliminary analysis suggests there are no funnels.

This hash was not in the original Dr. Dobb's article. I implemented it to fill a set of requirements posed by Colin Plumb. Colin ended up using an even simpler (and weaker) hash that was sufficient for his purpose.

Bernstein's hash

```
ub4 bernstein(ub1 *key, ub4 len, ub4 level)
{
  ub4 hash = level;
  ub4 i;
  for (i=0; i<len; ++i) hash = 33*hash + key[i];
  return hash;
}
```

If your keys are lowercase English words, this will fit 6 characters into a 32-bit hash with no collisions (you'd have to compare all 32 bits). If your keys are mixed case English words, 65*hash+key[i] fits 5 characters into a 32-bit hash with no collisions. That means this type of hash can produce (for the right type of keys) fewer collisions than a hash that gives a more truly random distribution. If your platform doesn't have fast multiplies, no sweat, 33*hash = hash+(hash<<5) and most compilers will figure that out for you.

On the down side, if you don't have short text keys, this hash has a easily detectable flaws. For example, there's a 3-into-2 funnel that 0x0021 and 0x0100 both have the same hash (hex 0x21, decimal 33) (you saw that one coming, yes?).

FNV Hash

I need to fill this in. See it on wikipedia. It's faster than lookup3 on Intel (because Intel has fast multiplication), but slower on most other platforms. Preliminary tests suggested it has decent distributions. It's public domain.

I have three complaints against it. First, it's specific about how to reduce the size if you don't use all the bits, it's not just a mask. Increasing the result size by one bit gives you a completely different hash. If you use a hash table that grows by increasing the result size by one bit, one old bucket maps across the entire new table, not to just two new buckets. If your algorithm has a sliding pointer for which buckets have been split, that just won't work with FNV. Second, it's linear. That means that widely separated things can cancel each other out at least as easily as nearby things. Third, since multiplication only affects higher bits, the lowest 7 bits of the state are never affected by the 8th bit of all the input bytes.

On the plus side, very nearby things never cancel each other out at all. This makes FNV a good choice for hashing very short keys (like single English words). FNV is more robust than Bernstein's hash. It can handle any byte values, not just ASCII characters.

```
ub4 fnv()
{
  /* I need to fill this in */
}
```

Goulburn Hash

The Goulburn hash is like my one-at-a-time, but more thorough and slower for all lengths beyond 0, asymptotically over 2x slower. It has two tables, g_table0 and g_table1, of respectively 256 and 128 4-byte integers.

For large hash tables (which is where being more thorough ought to buy you something), it does worse, because its internal operations not reversible. Specifically h^=rotate(h,3) and h^=rotate(h,14), which each cause an even number of bits to be set. I hashed the 2^{32} 32-bit integers with lookup3, one-at-a-time, and goulburn, and they produced 2,696,784,567, and 1,667,635,157, and 897,563,758 values respectively. The expected number for a random mapping would be 2,714,937,129 values.

```
u4 goulburn( const unsigned char *cp, size_t len, uint32_t last_value)
{
  register u4 h = last_value;
  int u;
  for( u=0; u<len; ++u ) {
    h += g_table0[ cp[u] ];
    h ^= (h << 3) ^ (h >> 29);
    h += g_table1[ h >> 25 ];
    h ^= (h << 14) ^ (h >> 18);
    h += 1783936964UL;
  }
  return h;
}
```

MurmurHash

I need to fill this in too. This is faster than any of my hash, and is more nonlinear than a rotating hash or FNV. I can see it's weaker than my lookup3, but I don't by how much, I haven't tested it.

http://murmurhash.googlepages.com/.

Cessu

The Cessu hash (search for msse2 in his blog) uses SSE2, allowing it to be faster and more thorough (at first glance) than what I can do 32 bits at a time. I haven't done more than a first glance at it.

Pearson's Hash

```
char pearson(char *key, ub4 len, char tab[256])
{
  char hash;
  ub4  i;
  for (hash=len, i=0; i<len; ++i)
    hash=tab[hash^key[i]];
  return (hash);
}
```

This preinitializes `tab[]` to an arbitrary permutation of 0 .. 255. It takes 6*n*+2 instructions, but produces only a 1-byte result. Larger results can be made by running it several times with different initial hash values.

CRC Hashing

```
ub4 crc(char *key, ub4 len, ub4 mask, ub4 tab[256])
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ tab[(hash & 0xff) ^ key[i]];
  return (hash & mask);
}
```

This takes 9*n*+3 instructions. `tab` is initialized to simulate a maximal-length Linear Feedback Shift Register (LFSR) which shifts out the low-order bit and XORs with a polynomial if that bit was 1. I used a 32-bit state with a polynomial of `0xedb88320` for the tests. Keys that differ in only four consecutive bytes will not collide.

A sample implementation, complete with table, is here.

You could also implement it like

```
ub4 crc(char *key, ub4 len, ub4 mask, ub4 tab[256])
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash << 8) ^ tab[(hash >> 24) ^ key[i]];
  return (hash & mask);
}
```

but, since shifts are sometimes slow, the other way might be faster. If you did it that way you'd have to reverse the bits of the generating polynomial because bits shift out the top instead of the bottom.

Generalized CRC Hashing

This is exactly the same code as CRC hashing except it fills tab[] with each of the 4 bytes forming a random permutation of 0..255. Unlike a true CRC hash, its mixing is nonlinear. Keys that differ in only one byte will not collide. The top byte has to be a permutation of 0..255 so no information is lost when the low byte is shifted out. The other bytes are permutations of 0..255 only to make hold the guarantee that keys differing in one byte will not collide.

A sample implementation, complete with table, is here.

Universal Hashing

```
ub4 universal(char *key, ub4 len, ub4 mask, ub4 tab[MAXBITS])
{
  ub4 hash, i;
  for (hash=len, i=0; i<(len<<3); i+=8)
  {
    register char k = key[i>>3];
    if (k&0x01) hash ^= tab[i+0];
    if (k&0x02) hash ^= tab[i+1];
    if (k&0x04) hash ^= tab[i+2];
    if (k&0x08) hash ^= tab[i+3];
    if (k&0x10) hash ^= tab[i+4];
    if (k&0x20) hash ^= tab[i+5];
    if (k&0x40) hash ^= tab[i+6];
    if (k&0x80) hash ^= tab[i+7];
  }
  return (hash & mask);
}
```

This takes 52*n*+3 instructions. The size of tab[] is the maximum number of input bits. Values in tab[] are chosen at random. Universal hashing can be implemented faster by a Zobrist hash with carefully chosen table values.

Zobrist Hashing

```
ub4 zobrist( char *key, ub4 len, ub4 mask, ub4 tab[MAXBYTES][256])
{
  ub4 hash, i;
  for (hash=len, i=0; i<len; ++i)
    hash ^= tab[i][key[i]];
  return (hash & mask);
}
```

This takes 10*n*+3 instructions. The size of tab[][256] is the maximum number of input bytes. Values of tab[][256] are chosen at random. This can implement universal hashing, but is more general than universal hashing.

Zobrist hashes are especially favored for chess, checkers, othello, and other situations where you have the hash for one state and you want to compute the hash for a closely related state. You xor to the old hash the table values that you're removing from the state, then xor the table values that you're adding. For chess, for example, that's 2 xors to get the hash for the next position given the hash of the current position.

Paul Hsieh's hash

This is kind of a cross between that big hash at the start of this article and my one-at-a-time hash. Paul's timed it and it was than that big hash. It has a 4-byte internal state that it does light nonlinear mixing after every combine. That's good. It combines 2-byte blocks with its 4-byte state, which is something I'd never tried. (FNV and CRC and one-at-a-time combine 1-byte blocks with the 4-byte state. Their input blocks are all smaller than their state, and they mix their state after each input block, which makes it impossible for consecutive input blocks to cancel.)

On the down side, it has funnels of 3 bits into 2, for example hex 01 00 00 00 00 00 00 00 and 00 00 20 00 01 00 00 00 both hash to 0xc754ae23.

```
#include "pstdint.h" /* Replace with  if appropriate */
#undef get16bits
#if (defined(__GNUC__) && defined(__i386__)) || defined(__WATCOMC__) \
  || defined(_MSC_VER) || defined (__BORLANDC__) || defined (__TURBOC__)
#define get16bits(d) (*((const uint16_t *) (d)))
#endif

#if !defined (get16bits)
#define get16bits(d) ((((const uint8_t *)(d))[1] << UINT32_C(8))\
                      +((const uint8_t *)(d))[0])
#endif

uint32_t SuperFastHash (const char * data, int len) {
uint32_t hash = len, tmp;
int rem;

    if (len <= 0 || data == NULL) return 0;
    rem = len & 3;
    len >>= 2;

    /* Main loop */
    for (;len > 0; len--) {
        hash  += get16bits (data);
        tmp    = (get16bits (data+2) << 11) ^ hash;
        hash   = (hash << 16) ^ tmp;
        data  += 2*sizeof (uint16_t);
        hash  += hash >> 11;
    }

    /* Handle end cases */
    switch (rem) {
        case 3: hash += get16bits (data);
                hash ^= hash << 16;
                hash ^= data[sizeof (uint16_t)] << 18;
                hash += hash >> 11;
                break;
        case 2: hash += get16bits (data);
                hash ^= hash << 11;
                hash += hash >> 17;
                break;
        case 1: hash += *data;
                hash ^= hash << 10;
                hash += hash >> 1;
    }

    /* Force "avalanching" of final 127 bits */
    hash ^= hash << 3;
    hash += hash >> 5;
    hash ^= hash << 4;
    hash += hash >> 17;
    hash ^= hash << 25;
    hash += hash >> 6;

    return hash;
}
```

My Hash

This takes 6*n*+35 instructions. It's the big one at the beginning of the article. It's implemented along with a self-test at http://burtleburtle.net/bob/c/lookup2.c.

lookup3.c

A hash I wrote nine years later designed along the same lines as "My Hash", see http://burtleburtle.net/bob/c/lookup3.c. It takes 2*n* instructions per byte for mixing instead of 3*n*. When fitting bytes into registers (the other 3*n* instructions), it takes advantage of alignment when it can (a trick learned from Paul Hsieh's hash). It doesn't bother to reserve a byte for the length. That allows zero-length strings to require no mixing. More generally, the length that requires additional mixes is now 13-25-37 instead of 12-24-36.

One theoretical insight was that the last mix doesn't need to do well in reverse (though it has to affect all output bits). And the middle mixing steps don't have to affect all output bits (affecting some 32 bits is enough), though it does have to do well in reverse. So it uses different mixes for those two cases. "My Hash" (lookup2.c) had a single mixing operation that had to satisfy both sets of requirements, which is why it was slower.

On a Pentium 4 with gcc 3.4.?, Paul's hash was usually faster than lookup3.c. On a Pentium 4 with gcc 3.2.?, they were about the same speed. On a Pentium 4 with icc -O2, lookup3.c was a little faster than Paul's hash. I don't know how it would play out on other chips and other compilers. lookup3.c is slower than the additive hash pretty much forever, but it's faster than the rotating hash for keys longer than 5 bytes.

lookup3.c does a much more thorough job of mixing than any of my previous hashes (lookup2.c, lookup.c, One-at-a-time). All my previous hashes did a more thorough job of mixing than Paul Hsieh's hash. Paul's hash does a good enough job of mixing for most practical purposes.

The most evil set of keys I know of are sets of keys that are all the same length, with all bytes zero, except with a few bits set. This is tested by frog.c.. To be even more evil, I had my hashes return b and c instead of just c, yielding a 64-bit hash value. Both lookup.c and lookup2.c start seeing collisions after 2^{53} frog.c keypairs. Paul Hsieh's hash sees collisions after 2^{17} keypairs, even if we take two hashes with different seeds. lookup3.c is the only one of the batch that passes this test. It gets its first collision somewhere beyond 2^{63} keypairs, which is exactly what you'd expect from a completely random mapping to 64-bit values.

MD4

This takes 9.5*n*+230 instructions. MD4 is a hash designed for cryptography by Ron Rivest. It takes 420 instructions to hash a block of 64 aligned bytes. I combined that with my hash's method of putting unaligned bytes into registers, adding 3*n* instructions. MD4 is overkill for hash table lookup.

The table below compares all these hash functions.

NAME
: is the name of the hash.

SIZE-1000
: is the smallest reasonable hash table size greater than 1000.

SPEED
: is the speed of the hash, measured in instructions required to produce a hash value for a table with SIZE-1000 buckets. It is assumed the machine has a rotate instruction. These aren't very accurate measures ... I should really just do timings on a Pentium 4 or such.

INLINE
: This is the speed assuming the hash is inlined in a loop that has to walk through all the characters anyways, such as a tokenizer. Such a loop doesn't always exist, and even when it does inlining isn't always possible. Some hashes (my new hash and MD4) work on blocks larger than a character. Inlining a hash removes 3*n*+1 instructions of loop overhead. It also removes the *n* instructions needed to get the characters out of the key array and into a register. It also means the length isn't known. Inlining offers other advantages. It allows the string to be converted to uppercase, and/or to unicode, before the hash is performed without the expense of an extra loop or a temporary buffer.

FUNNEL-15
: is the largest set of input bits affecting the smallest set of internal state bits when mapping 15-byte keys into a 1-byte result.

FUNNEL-100
: is the largest set of input bits affecting the smallest set of internal state bits when mapping 100-byte keys into a 32-bit result.

COLLIDE-32
: is the number of collisions found when a dictionary of 38,470 English words was hashed into a 32-bit result. (The expected number of collisions is 0.2 .)

COLLIDE-1000
: is a chi^2 measure of how well the hash did at mapping the 38470-word dictionary into the SIZE-1000 table. (A chi^2 measure greater than +3 is significantly worse than a random mapping; less than -3 is significantly better than a random mapping; in between is just random fluctuations.)

Comparison of several hash functions

NAME	SIZE-1000	SPEED	INLINE	FUNNEL-15	FUNNEL-100	COLLIDE-32	COLLIDE-1000
Additive	1009	5*n*+3	*n*+2	15 into 2	100 into 2	37006	+806.02
Rotating	1009	6*n*+3	2*n*+2	4 into 1	25 into 1	24	+1.24
One-at-a-Time	1024	9*n*+9	5*n*+8	none	none	0	-0.05

Bernstein	1024	7*n*+3	3*n*+2	3 into 2	3 into 2	4	+1.69
FNV	1024	?	?	?	?	?	?
Pearson	1024	12*n*+5	4*n*+3	none	none	0	+1.65
CRC	1024	9*n*+3	5*n*+2	2 into 1	11 into 10	0	+0.07
Generalized	1024	9*n*+3	5*n*+2	none	none	0	-1.83
Universal	1024	52*n*+3	48*n*+2	4 into 3	50 into 28	0	+0.20
Zobrist	1024	10*n*+3	6*n*+2	none	none	1	-0.03
Paul Hsieh's	1024	5*n*+17	N/A	3 into 2	3 into 2	1	+1.12
My Hash	1024	6*n*+35	N/A	none	none	0	+0.33
lookup3.c	1024	5*n*+20	N/A	none	none	0	-0.08
MD4	1024	9.5*n*+230	N/A	none	none	1	+0.73

From the measurements we can conclude that the Additive and Rotating hash (and maybe Bernstein) were noticably bad for 32-bit results, and only the Additive hash was noticably bad for 10-bit results. If inlining is possible, the Rotating hash was the fastest acceptable hash, followed by Bernstein, Pearson or the Generalized CRC (if table lookup is OK) or Bernstein or One-at-a-Time (if table lookup is not OK). If inlining is not possible, it's a draw between lookup3 and Paul Hsieh's hash. Note that, for many applications, the Rotating hash is noticably bad and should not be used, and the Bernstein hash is marginal. Table lengths should always be a power of 2 because that's faster than prime lengths and all acceptable hashes allow it.

The COLLIDE-1000 numbers should be ignored, unless the numbers are bigger than 3 or less than -3. For example, generalized CRC produced +.8, -.8, or -1.8 for three different tables I tried. It's just noise. A different set of keys would give unrelated random fluctuations.

Conclusion

A common weakness in hash function is for a small set of input bits to cancel each other out. There is an efficient test to detect most such weaknesses, and many functions pass this test. I gave code for the fastest such function I could find. Hash functions without this weakness work equally well on all classes of keys.

Testimonials:

- A fractal mountain and image of same