FNV hash history

The basis of the **FNV** hash algorithm was taken from an idea sent as reviewer comments to the IEEE POSIX P1003.2 committee by Glenn Fowler and Phong Vo. In a subsequent ballot round: Landon Curt Noll improved on their algorithm. Some people tried this hash and found that it worked rather well. In an EMail message to Landon, they named it the ``**Fowler/Noll/Vo**'' or **FNV** hash.

FNV hashes are designed to be fast while maintaining a low collision rate. The **FNV** speed allows one to quickly hash lots of data while maintaining a reasonable collision rate. The high dispersion of the **FNV** hashes makes them well suited for hashing nearly identical strings such as URLs, hostnames, filenames, text, IP addresses, etc.

The **FNV** hash is in wide spread use:

- calc
- Domain Name Servers
- mdbm key/value data lookup functions
- Database indexing hashes
- major web search / indexing engines
- high performance EMail servers
- Netnews history file Message-ID lookup functions
- Anti-spam filters
- NFS implementations (e.g., FreeBSD 4.3, IRIX, Linux)

FNV hash algorithms and source code have been released into the public domain.

If you use an **FNV** function in an application, tells us about it by sending EMail to:
fnv-mail@asthe.com

We will be happy to add your application to the list.

Comments are welcome.

The core of the FNV hash

The core of the **FNV-1** hash algorithm is as follows:

```
hash = offset_basis
for each octet_of_data to be hashed
        hash = hash * FNV_prime
        hash = hash xor octet_of_data
return hash
```

The ***offset_basis*** and ***FNV_prime*** can be found in the parameters of the FNV-1 hash section below.

FNV-1a alternate algorithm

There is a minor variation of the **FNV** hash algorithm known as **FNV-1a**:

```
hash = offset_basis
for each octet_of_data to be hashed
        hash = hash xor octet_of_data
        hash = hash * FNV_prime
return hash
```

The only difference between the **FNV-1a** hash and the **FNV-1** hash is the order of the xor and multiply. The **FNV-1a** hash uses the same ***FNV_prime*** and ***offset_basis*** as the **FNV-1** hash of the same **n**-bit size.

Some people use **FNV-1a** instead of **FNV-1** because they see slightly better dispersion for tiny (<4 octets) chunks of memory.

Either **FNV-1** or **FNV-1a** make a fine hash. (Try it with with just a dash of Sage and ground Cloves :-))

Parameters of the FNV-1 hash

- **hash** is an **n** bit unsigned integer, where **n** is the bit length of **hash**.

- The multiplication is performed modulo 2^n where **n** is the bit length of **hash**.
- The xor is performed on the low order octet (8 bits) of **hash**.
- The ***FNV_prime*** is dependent on **n**, the size of the hash:

  32 bit ***FNV_prime*** = `16777619`
  64 bit ***FNV_prime*** = `1099511628211`
  128 bit ***FNV_prime*** = `309485009821345068724781401`
  256 bit ***FNV_prime*** = `374144419156711147060143317175368453031918731002211`

  Part of the magic of **FNV** is the selection of the ***FNV_prime*** for a given sized unsigned integer. Some primes do hash better than other primes for a given integer size. The theory behind which primes make good ***FNV_prime***'s is beyond the scope of this web page.

- The ***offset_basis*** for **FNV-1** is dependent on **n**, the size of the hash:

  32 bit ***offset_basis*** = `2166136261`
  64 bit ***offset_basis*** = `14695981039346656037`
  128 bit ***offset_basis*** = `275519064689413815358837431229664493455`
  256 bit ***offset_basis*** =
  `100029257958052580907070968620625704837092796014241193945225284501741471925557`

  These non-zero integers are the FNV-0 hashes of the following 32 octets:

  ```
  chongo <Landon Curt Noll> /\../\
  ```

  The \'s in the above string are not C-style escape characters. In C-string notation, these 32 octets are:

  ```
  "chongo <Landon Curt Noll> /\\../\\"
  ```

  The following calc script was used to compute the ***offset_basis*** for **FNV-1** hashes:

  ```
  offset_basis = 0;
  FNV_prime = insert_the_FNV_prime_here;
  hash_bits = insert_the_hash_size_in_bits_here;
  offset_str = "chongo <Landon Curt Noll> /\\../\\";
  hash_mod = 2^hash_bits;

  str_len = strlen(offset_str);
  for (i=1; i <= str_len; ++i) {
      offset_basis = (offset_basis * FNV_prime) % hash_mod;
      offset_basis = xor(offset_basis, ord(substr(offset_str,i,1)));
  }

  print hash_bits, "bit offset_basis =", offset_basis;
  ```

  NOTE: The above code fragment example is written in the calc language, not in C.

  FNV-0 Historic note: The **FNV-0** is the historic **FNV** algorithm that is now deprecated. It has an ***offset_basis*** of `0`. Unless the **FNV-0** hash is required for historical purposes, the **FNV-1** should be used in place of the **FNV-0** hash. Use **FNV-1** with its non-zero ***offset_basis*** instead. The **FNV-0** hashes all buffers that contain only 0 octets to a hash value of 0. The **FNV-1** does not suffer from this minor problem.

Changing the FNV hash size - xor-folding

If you need an **x**-bit hash where **x** is not a power of 2, then we recommend that you compute the **FNV** hash that is just larger than **x**-bits and *xor-fold* the result down to **x**-bits. By *xor-folding* we mean shift the excess high order bits down and xor them with the lower **x**-bits. For example to produce a 24 bit **FNV-1** hash in C we *xor-fold* fold a 32 bit **FNV-1** hash:

```
#define MASK_24 (((u_int32_t)1<<24)-1)      /* i.e., (u_int32_t)0xffffff */
#define FNV1_32_INIT ((u_int32_t)2166136261)
u_int32_t hash;
void *data;
size_t data_len;

hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
hash = (hash>>24) ^ (hash & MASK_24);
```

To produce a 16 bit **FNV-1** hash in C we *xor-fold* fold a 32 bit **FNV-1** hash:

```
#define MASK_16 (((u_int32_t)1<<16)-1)      /* i.e., (u_int32_t)0xffff */
#define FNV1_32_INIT ((u_int32_t)2166136261)
u_int32_t hash;
void *data;
size_t data_len;

hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
hash = (hash>>16) ^ (hash & MASK_16);
```

To produce a 56 bit **FNV-1** hash in C (on a machine with 64 bit unsigned values) we *xor-fold* fold a 64 bit **FNV-1** hash:

```
#define MASK_56 (((u_int64_t)1<<56)-1)      /* i.e., (u_int64_t)0xffffffffffffff */
#define FNV1_64_INIT ((u_int64_t)14695981039346656037)
u_int64_t hash;
void *data;
size_t data_len;

hash = fnv_64_buf(data, data_len, FNV1_64_INIT);
hash = (hash>>56) ^ (hash & MASK_56);
```

If you really need an **n**-bit hash for **n** > 256 bits, send us EMail.

Changing the FNV hash size - non-powers of 2

The FNV hash is designed for hash sizes that are a power of 2. If you need a hash size that is not a power of two, then you have two choices. One method id called the ***lazy mod mapping method*** and the other is called the ***retry method***. Both involve mapping a range that is a power of 2 onto an arbitrary range.

- ***Lazy mod mapping method***: The ***lazy mod mapping method*** uses a simple mod on an **n**-bit hash to yield an arbitrary range. To produce a hash range between **0** and **X** use a **n**-bit FNV hash where **n** is smallest FNV hash that will produce values larger than **X** without the need for xor-folding.

For example, to produce a value between **0** and **2142779559** using the ***lazy mod mapping method***, we select a **32**-bit FNV hash because:

$2^{32} >$ **2142779559**

We compute the **32**-bit FNV hash value and then perform a final mod:

```
#define TRUE_HASH_SIZE ((u_int32_t)2142779560) /* range top plus 1 */
#define FNV1_32_INIT ((u_int32_t)2166136261)
u_int32_t hash;
void *data;
size_t data_len;

hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
hash %= TRUE_HASH_SIZE;
```

An advantage of the ***lazy mod mapping method*** is that it requires only 1 more operation: only an additional mod is performed at the end. The disadvantage of the ***lazy mod mapping method*** is that there is a bias against the larger values.

To understand this bias consider the a need to produce a value between **0** and **999999**. We will compute a **32**-bit FNV hash value because:

$2^{32} >$ **999999**

We compute the **32**-bit FNV hash value using the and then perform the final mod:

```
#define TRUE_HASH_SIZE ((u_int32_t)1000000) /* range top plus 1 */
#define FNV1_32_INIT ((u_int32_t)2166136261)
u_int32_t hash;
void *data;
size_t data_len;

hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
hash %= TRUE_HASH_SIZE;
```

The bias introduced by the final mod is slight. The values **0** through **967295** will be created by 4295 different **32**-bit FNV hash values whereas the values **967296** through **999999** will be created by only 4294 different **32**-bit FNV hash values. In other words, the values **0** through **967295** will occur ~1.0002328 times as often as the values **967296** through **999999**.

The bias can be larger when the range is nearly as large as the range of values produced by the FNV hash. Consider using the ***lazy mod mapping method*** to produce values between **0** and **9999999999999999999**. We use a **64**-bit FNV hash because:

$2^{64} >$ **9999999999999999999**

We compute the **64**-bit FNV hash value using the and then perform the final mod:

```
#define TRUE_HASH_SIZE ((u_int64_t)10000000000000000000) /* range top plus 1 */
#define FNV1_64_INIT ((u_int64_t)14695981039346656037)
u_int64_t hash;
void *data;
size_t data_len;

hash = fnv_64_buf(data, data_len, FNV1_64_INIT);
hash %= TRUE_HASH_SIZE;
```

Here the bias introduced by the final mod is more noticeable. The values **0** through **9999999999999999999** will be created by 2 different **64**-bit FNV hash values whereas the values **10000000000000000000** through **18446744073709551615** will be created by only 1 **64**-bit FNV hash value.

NOTE: This bias issue may not be of concern to you, but we thought we should point out this issue just in case you care. Most of the time applications and people need / should / will not care about this bias.

- ***Retry method***: The ***retry method*** also performs a final mod in order to produce a hash range between **0** and **X**. Unlike ***lazy mod mapping method***, the ***retry method*** avoids the bias by additional computation.

  To produce a hash range between **0** and **X** use a **n**-bit FNV hash where **n** is smallest FNV hash that will produce values larger than **X** without the need for xor-folding.

  For example, to produce a value between **0** and **49999** using the ***retry method***, we select a **32**-bit FNV hash because:

  $2^{32} >$ **49999**

  Before the final mod **50000** is performed, we check to see if the **32**-bit FNV hash value is one of the upper biased values. If it is, we perform additional loop cycles until is below the bias level. For example:

```
#define TRUE_HASH_SIZE ((u_int32_t)50000) /* range top plus 1 */
#define FNV_32_PRIME ((u_int32_t)16777619)
#define FNV1_32_INIT ((u_int32_t)2166136261)
#define MAX_32BIT ((u_int32_t)0xffffffff) /* largest 32 bit unsigned value */
#define RETRY_LEVEL ((MAX_32BIT / TRUE_HASH_SIZE) * TRUE_HASH_SIZE)
u_int32_t hash;
void *data;
size_t data_len;

hash = fnv_32_buf(data, data_len, FNV1_32_INIT);
while (hash >= RETRY_LEVEL) {
    hash = (hash * FNV_32_PRIME) + FNV1_32_INIT;
}
hash %= TRUE_HASH_SIZE;
```

  The disadvantage of the ***retry method*** is that it sometimes requires additional calculations. An advantage of the ***retry method*** it avoids slightly biased values.

  For another example, we will produce a value between **0** and **999999999999** using the ***retry method***, we select a **64**-bit FNV hash because:

  $2^{64} >$ **999999999999**

  Before the final mod **1000000000000** is performed, we check to see if the **64**-bit FNV hash value is one of the upper biased value. If it is, we perform additional loop cycles until it is not.

```
#define TRUE_HASH_SIZE ((u_int64_t)1000000000000) /* range top plus 1 */
#define FNV_64_PRIME ((u_int64_t)1099511628211)
#define FNV1_64_INIT ((u_int64_t)14695981039346656037)
#define MAX_64BIT ((u_int64_t)0xffffffffffffffff) /* largest 64 bit unsigned value */
#define RETRY_LEVEL ((MAX_64BIT / TRUE_HASH_SIZE) * TRUE_HASH_SIZE)
u_int64_t hash;
void *data;
size_t data_len;

hash = fnv_64_buf(data, data_len, FNV1_64_INIT);
```

```
while (hash >= RETRY_LEVEL) {
    hash = (hash * FNV_64_PRIME) + FNV1_64_INIT;
}
hash %= TRUE_HASH_SIZE;
```

- ***To summarize***: When dealing with an application that needs to generate a hash value over an arbitrary range, one can do one of the following:

  1. Change the application to use hash values that range between **0** and $\mathbf{2^n-1}$. Use a **n**-bit FNV hash, xor-folding if needed.

     Pro: Yields the best results in the shortest amount of CPU time.
     Con: Requires source code change to force hash range to be a power of 2 in size.

  2. Use the ***lazy mod mapping method*** if one does not care about the slight hash bias and does not want (or cannot change) the hash range.

     Pro: Yields the fastest results for a non-power of 2 range.
     Con: Produces a slight bias against larger hash values. However if one does not care about the slight bias, then there is no problem using this technique.

  3. Use the ***retry method*** if one wants to avoid the hash bias and does not want / cannot change the hash range.

     Pro: Produces non-biased values for a non-power of 2 range.
     Con: Requires slightly more CPU time in some cases.

FNV source

In the C FNV source below, primes are provided for 32 bit and 64 bit unsigned integers. For compilers that do not implement the *unsigned long long* type, code is provided to quickly simulate the 64 bit multiply by the particular ***FNV_prime***.

- fnv-4.1.tar.gz - (all the bits)

- hash_32.c - (32 bit **FNV-1** algorithm)
- hash_64.c - (64 bit **FNV-1** algorithm)
- hash_32a.c - (32 bit **FNV-1a** algorithm)
- hash_64a.c - (64 bit **FNV-1a** algorithm)
- fnv.h - (**FNV** header file)

- fnv32.c - (32 bit **FNV-0** and **FNV-1** hash tool/demo)
- fnv64.c - (64 bit **FNV-0** and **FNV-1** hash tool/demo)
- fnv32a.c - (32 bit **FNV-1a** hash tool/demo)
- fnv64a.c - (64 bit **FNV-1a** hash tool/demo)

- README - (brief comments about **FNV-0** and **FNV-1**)
- Makefile - (how to compile/install)
- have_ulong64.c - (64 bit unsigned integer type detector)

gcc optimization

It has been reported by several people that under the gcc compiler with -O3 on many AMD & Intel CPUs, that replacing the *FNV_prime* multiply with a expression of shifts and adds will improve the performance.

Limited testing on our part confirmed that one can gain a few % in speed on an 1.6GHz AMD Athlon using gcc version 3.2.2 with -O3 optimization.

For a 32 bit FNV-1, we used:

```
while (bp < be) {

    /* multiply by the 32 bit FNV magic prime mod 2^32 */
#if defined(NO_FNV_GCC_OPTIMIZATION)
    hval *= FNV_32_PRIME;
#else
    hval += (hval<<1) + (hval<<4) + (hval<<7) + (hval<<8) + (hval<<24);
#endif

    /* xor the bottom with the current octet */
    hval ^= (Fnv32_t)*bp++;
}
```

For a 32 bit FNV-1a, we used:

```
while (bp < be) {

    /* xor the bottom with the current octet */
    hval ^= (Fnv32_t)*bp++;

    /* multiply by the 32 bit FNV magic prime mod 2^32 */
#if defined(NO_FNV_GCC_OPTIMIZATION)
    hval *= FNV_32_PRIME;
#else
    hval += (hval<<1) + (hval<<4) + (hval<<7) + (hval<<8) + (hval<<24);
#endif
}
```

For a 64 bit FNV-1, we used:

```
while (bp < be) {

    /* multiply by the 64 bit FNV magic prime mod 2^64 */
#if defined(NO_FNV_GCC_OPTIMIZATION)
    hval *= FNV_64_PRIME;
#else
    hval += (hval << 1) + (hval << 4) + (hval << 5) +
            hval << 7) + (hval << 8) + (hval << 40);
#endif

    /* xor the bottom with the current octet */
    hval ^= (Fnv64_t)*bp++;
}
```

For a 64 bit FNV-1a, we used:

```
while (bp < be) {

    /* xor the bottom with the current octet */
    hval ^= (Fnv64_t)*bp++;

    /* multiply by the 64 bit FNV magic prime mod 2^64 */
#if defined(NO_FNV_GCC_OPTIMIZATION)
    hval *= FNV_64_PRIME;
#else
    hval += (hval << 1) + (hval << 4) + (hval << 5) +
            hval << 7) + (hval << 8) + (hval << 40);
#endif
}
```

Landon Curt Noll
chongo <was here> **/\oo/**