Perfect(ing) hashing in NetBSD

Jörg Sonnenberger

March 16, 2013

Abstract

Hash tables are one of the fundamental data structures used in the different parts of an Operating System from on-disk databases to directory caches in the kernel. To provide the desired performance characteristics, keeping the collision rate minimal is crucial. If the construction of the hash function guarantees that no collisions occur, it is a Perfect Hash Function. This paper gives a short introduction into the algorithms developed in the last 20 years for this task. It looks at the uses in NetBSD 6 like *nbperf* and the constant database *cdb* as well as work-in-progress code for routing and firewalling.

1 Introduction

Hash tables are data structures used to implement associative arrays, i.e. they provide a mapping of arbitrary complex keys to values or indices. They are one of the oldest data structures for this purpose and since their discovery in 1953, they have been studied extensively and can be found in many programs and libraries. The wide spread use is a result of the SMP friendliness and efficiency as ideally, inserts, lookups and removals are all constant time operations.

The core of every hash table implementation is the hash function. Typical examples are using the remainder of the division by a prime or Bernstein's string hash $h(x[0..n]) = 33 \times h(x[0..n-1]) + x[n]$ with $h(0) = 5381$[1]. All simple hash functions share one important property: certain input keys are mapped to the same index. This is called a hash collision and requires special handling. Techniques for dealing with them include linked lists between elements with the same hash, repeating the hashing with an additional counter as argument to find a different position or resizing the hash table.

In the last years, a number of so-called complexity attacks[5] have been published where an attacker explicitly creates hash collisions to force the target to waste time by slowing down the operations from expected constant time to linear in the number of keys. One way to address the complexity attacks is to move from hash tables to balanced trees at the cost of logarithmic complexity for most operations. The other alternative is deploying randomised hash functions.

[1]This hash is also known as DJBX33A.

Randomised hash functions are also the building block for more powerful hash table schemes: perfect hash functions. A Perfect Hash Function (PHF) is constructed in such a way, that any two keys of a known input set are mapped to different indices, i.e. that no hash collisions can exist. This makes them a perfect match for applications with a (mostly) fixed key set. If the PHF also maps the n keys to $0..n-1$, it is called a Minimal Perfect Hash Function (MPHF). The most well known program for creating PHF and MPHF is GNU gperf. It has been used for the keyword recognition in compilers like GCC.

This paper introduces the important developments in this area of research since 1990. As practical applications the *nbperf* program in NetBSD 6 is presented as well the new constant database *cdb*. An outlook to further work for using perfect hashing in the routing table and in NPF is also presented.

2 New algorithms for Perfect Hash Functions

This section looks at the early algorithms for PHF construction and challenges faced. It introduces the most noticable modern algorithms developed since 1990 and how they work.

The best algorithmic choice for a specific application depends on a number of common factors:

- Does the hash function preserve the key order? If it does, integration is easier as the existing table structures can be reused and the hash function works as additional secondary index.
- How much space does the hash function needs per key?
- What computations are needed for the hash function?
- If the algorithm constructs a non-minimal PHF, what key density can it achieve?

Comparing the individual properties and weighting them according the specific needs results in the correct choice.

2.1 Systematic construction of Perfect Hash Functions until 1990

Different authors have investigated construction mechanisms for Perfect Hash Functions since the invention of the hash table. Bostic published the predecessor of GNU gperf around 1984. Knuth discussed examples in The Art Of Computer Programming. A beautiful construction can be found in Pearson's paper "Fast Hashing of Variable-Length Text Strings" from 1990[7].

It is based on an 8-bit permutation table and traversed according to the XOR combination of the last hash value and the current input character. The design looks very similiar to the RC4 stream cipher. Pearson's paper explicitly discusses ways to systematically search for a permutation, but also the limitations. A trivial case of a problematic input is given where the target range has to be shifted to produce a MPHF and it can be easily seen that the algorithm doesn't scale very well with the number of keys.

The (lack of) scalability of the construction mechanism is a fundamental issue of the early approaches. If they work, they tend to provide very fast and moderately compact hash functions. For key sets larger than a few dozen keys at most, the construction will generally fail or require exponential construction time. For this reason, perfect hashing hasn't been deployed but for compiler construction for a long time.

2.2 Czech, Havas and Majewski's Minimal Perfect Hash Functions

The CHM construction was published in 1992[2]. It is one of the earliest, if not the earliest, expected linear time algorithm for the construction of MPHFs. Expected linear time in this case means that the algorithm uses randomised graphs with certain properties and try again, if a specific choice doesn't fit. Each run of the algorithm takes linear time and the chance of requiring more than one run is very low.

The resulting hash function has the useful property of being order preserving. This means that the hash function preserves the input order by mapping the n-th key is mapped to $n-1$. In practise this makes the hash function very easy to fit into existing code as e.g. tables mapping actions to handlers don't have to be reordered to fit the hash.

The algorithm depends one two central concepts. The first concept is the creation of a random graph by using two or more independent random hash functions. This graph has n edges and $m = cn$ vertices (for some $c \geq 1$). Each edge is created by taking the value of the chosen hash functions modulo m. If the graph is acyclic, the algorithm continues. Otherwise, another try is made with a different choice of random hash functions. If the constant c is chosen correctly, the graph is acyclic with a very high probability. When using two hash functions, c must be at least 2. When using three hash functions, c must be at least 1.24.

The second concept by Czech et al. is to continue by assigning a number to every vertex, so that the key number corresponding to each edge is the sum of the vertices of the edge modulo m. The initial value of all vertices is 0. An edge is chosen, so that one of the vertices has a degree of one. Without such an edge, the graph would contain a cyclic. Subsequently the value of the other vertex is updated, so that the edge fulfills the desired sum and the edge is removed afterwards.

The result requires storing m integers between 0 and $n-1$ as well as the parameters for the chosen random hash functions. It is the best known construction for order-preserving MPHF. The result hash function takes the compution of the two (three) chosen random hash functions, a modulo operation for each, two (three) table lookups, summing up the results and computing another modulo. The modulo operation itself can be replaced by two multiplications by computing the inverse. The storage requirement is at least $m \log_2 n$, but typically $m\lceil \log_2 n \rceil$ bits per key.

2.3 Botelho, Pagh and Ziviani's Perfect Hash Functions

The BPZ construction was published in 2007[1] and is very similar to the CHM algorithm. The main difference is the way numbers are assigned to the vertices. For BPZ, each edge is represented by one of its vertices. The sum of the corresponding numbers modulo 2 (3) gives the representative. As such the resulting hash function is not minimal by itself. At the same time, it requires much less space. When using three independent random hash functions, a value between 0 and 2 must be stored for all m vertices. One simple encoding stores five such values per byte ($3^5 = 243 < 256$). Using $c = 1.24$, this requires $1.24 \times n \times 8/5$ 1.98 bit storage per key with 24entries.

To obtain a MPHF, some post-processing is needed. The PHF can be reduced to a MPHF using a counting function. This function returns for a given index k how many "holes" the PHF has until k. This can be represented as bit vector with partial results ever so often memorised to keep the O(1) computation time. Careful choices require two additional table lookups and one 64 bit population count with a storage requirement of approximatily

2.79 bits per key using three random hash functions.

2.4 Belazzougui, Botelho and Dietzfelbinger's Perfect Hash Functions

"Hash, displace, and compress" or sort CHD was published in 2009[3] and is currently the algorithm known to create the smallest PHF. Using three random hash functions, a PHF can be constructed with 1.4 bit storage per key. The construction itself and the resulting hash function is a lot more complicated than BPZ though, so no further details will be provided.

3 nbperf

nbperf was started in 2009 to provide a replacement for GNU gperf that can deal with large key sets. cmph [2] was investigated for this purpose, but both the license and the implementation didn't fit into the NetBSD world. *nbperf* currently implements the CHM algorithm for 2-graphs and 3-graphs as well as the BPZ algorithm for 3-graphs. CHD wasn't available when the work on *nbperf* started and hasn't been implemented yet.

The output of *nbperf* is a single function that maps a pointer and size to the potential table entry. It does not validate the entry to avoid duplicating the keys or adding requirements on the data layout. If the callee of the hash function already knows that it has a valid key, it would also add overhead for no reason.

The most important option for *nbperf* is the desired construction algorithm. This affects the size and performance of the hash function. Hash functions using CHM are much larger. The 2-graph version requires two memory accesses, the 3-graph version three. The actual cache foot print depends on the number of keys as the size of the entries in the internal data array depends on that. For BPZ, the entries are much smaller, but some additional overhead is needed to provide a minimal hash function. As mentioned earlier, CHM is easier to use in applications, since it preserves the key order.

The following test case uses the Webster's Second International dictionary as shipped with NetBSD and the shorter secondary word list. They contain 234977 and 76205 lines. Each line is interpreted as one input key. *nbperf* is run with the "-p" option to get repeatable results. The "tries" column lists the number of iterations the program needed to find a usable random graph.

Input	Algorithm	Tries	Run time in s
web2	CHM	1	0.58
	CHM3	39	0.85
	BPZ	11	0.51
web2a	CHM	12	0.35
	CHM3	7	0.17
	BPZ	18	0.16

The resulting code for CHM can be seen in listing 1

The "mi_vector_hash" function provides an endian-neutral version of the Jenkin's hash function[4]. The third argument is the chosen seed. The modulo operations are normally replaced by two multiplications by the compiler.

At the time of writing, two applications in NetBSD use *nbperf*. The first user was the new *terminfo* library in NetBSD 6 and uses it for the key word list of *tic*. The second example is *apropos*, which contains a stop word list (i.e. words to be filtered from the query). This list is indexed by a Perfect Hash Function.

Further work for *nbperf* includes investigating simpler random hash function families to provide results with performance characteristics similar to GNU gperf's hashes. An implementation of the CHD algorithm is also planned.

4 The NetBSD constant database

NetBSD used the Berkeley Database to provide indexed access for a number of performance sensitive interfaces. This includes lookups for user names, user IDs, services and devices. The Berkeley Database has a number of limitations for this applications, which opened up the question of how to address these:

- Lack of atomic transactions,
- Database size overhead,
- Code complexity,
- Userland caching on a per-application base,

The first item ensures that most use cases in the system deploy a copy-on-write scheme for the (rare) case of modifications. It also means that any program has to be able to regenerate the database content after a disk crash.

The second item matters for embedded systems as it limits what database can be shipped pre-built. The third item is somewhat related, if the disk image doesn't require write support, it still can't leave the relevant code out.

The last item increases the memory foot print and reduces sharing data. It also adds overhead for

[2]http://cmph.sourceforge.net

Listing 1: CHM example

```c
#include <stdlib.h>

uint32_t
hash(const void * __restrict key, size_t keylen)
{
  static const uint32_t g[469955] = {
    /* ... */
  };
  uint32_t h[3];

  mi_vector_hash(key, keylen, 0x00000000U, h);

  return (g[h[0] % 469955] + g[h[1] % 469955]) % 234977;
}
```

multi-threaded applications as the library has to avoid concurrent access to the internal block cache.

NetBSD has imported SQLite, which provides a higher level library including transaction support. This doesn't help with the items above, especially the third. A new library was created to complement SQLite: the constant database. This format provides a read-only storage layer with deterministic access time, lock-free operation and based on memory mapped files to fully utilize the kernel cache.

The constant database (CDB) consists of two parts. The first part is the value storage, allowing access to each record using the record number. It can be used to iterate over the content or to link entries together. The second part provides a Perfect Hash Function for additional key based access. The keys are not stored on disk, so the application is responsible for doing any validation. For most file formats, the key is part is of the record anyway and especially when using multiple keys for the same record, storing would increase the file size without justification. Key lookup requires one computation of mi_vector_hash for the given key and reading three locations in the on-disk hash description. Worst case is thus three page faults with loading the blocks from disk. That gives the index and one more access the actual data offset. The result is a pointer, size pair directly into the memory mapped area. Looking up the same key twice therefore doesn't result in any additional IO nor does it require any traps, unless the system is low on memory.

In terms of code complexity, the CDB reader adds about 1.6KB to libc on AMD64 and writer around 4.3KB. As the database files are often the same size or smaller than the corresponding text sources, dropping the text versions can result in an overall decrease in size.

For NetBSD 6 the device database, the service database and libterminfo use the new CDB format. The resulting databases are typically less than one fourth of the size of the corresponding Berkeley DB files. The creation time has also improved. Further work is required to convert the remaining users in libc, but also to provide access in other programs like Postfix.

5 Perfect hashing for the route lookup

Cleaning up the routing code and investigating new data structures and/or implementations is on-going work in NetBSD. David Young provided the first major part for this by isolating access to the radix tree and hiding it behind a clean interface. The pending work moves the preference selecting (i.e. which of two routes with the same netmask is chosen) and the special case of host routes out of the radix tree into the generic layer. This changes will allow replacing the old BSD radix tree with less generic, but faster code. It also makes it possible to switch the lookup data structure for the fast path.

The most interesting alternatives are compressed tries[6] (out of the scope of this paper) and multi-level hashing[8]. Multi-level hashing is based on the idea of performing the CIDR[3] lookup as binary search on the possible prefix lengths. For IPv4, this could mean starting with looking for /16 routes and depending on match or not, continue with /8 or /24 entries. This requires adding markers for more specific routes to direct the search.

Consider the following routing table:

[3]Classless Inter-Domain Routing

Destination network	Gateway
0/0	192.168.0.10
127/8	127.0.0.1
192.168.0/24	192.168.0.2
192.168.1/24	192.168.0.1
10/8	192.168.0.1
10.0.10/24	192.168.0.5
10.192/12	192.168.0.6
11.192/12	192.168.0.7

A search for 10.0.10.1 will start by looking for 10.0/16 in the hash table to be constructed. No such route exists, but the search has to continue with larger prefix length to find the correct entry 10.0.10/24. For this purpose, a marker has to be added with entry 10.0/16 and a reference to 10/8. The reference avoids the need for backtracking, i.e. when searching for 10.0.11.1. They can either reference the covering route or copy the corresponding gateway, depending on the granularity of traffic accounting. With the additional marker entries, the following content of the hash table is enough:

Destination network	Type	Data
0/0	GW	192.168.0.10
127/8	GW	127.0.0.1
192.168/16	R	0.0.0.0/0
192.168.0/24	GW	192.168.0.2
192.168.1/24	GW	192.168.0.1
10/8	GW	192.168.0.1
10.0/16	R	10/8
10.0.10/24	GW	192.168.0.5
10.192/12	GW	192.168.0.6
11/8	R	0/0
11.192/12	GW	192.168.0.7

For this specific case, three additional entries are enough as the marker for 10.192/12 is 10/8 and that's already present as route. Using perfect hashing ensures a predictable lookup cost as it limits the number of expensive memory accesses. Using the BPZ algorithm with a 2-graph and no post-filtering means a hash table utilisation of 50% and approximately 2 bit per key storage for the hash function itself. It is possible to use a single hash table for all entries or to use a /separate table for each prefix length. The latter allows using 64 bit per entry in case of IPv4 (32 bit network, 32 bit as the next-hop identifier) and between 64 bit and 160 bit for IPv6. Even for a core router in the Default Free Zone, 100,000 entries and more fit into the L3 cache of modern CPU.

The downside of using perfect hashing is the construction time. Investigations have to be performed on how critical the resulting update time is for busy routers.

Further optimisations can be deployed. The optimal branching is often not a static binary search, so storing hints for the next level to look at can be useful. Published research by Waldvogel et al. suggests that the average number of hash table probes can be much less than 2, when chosing the correct order. The lookup table itself can avoid redundant entries, i.e. if a more specific router and the immediate covered route have the same next-hop. This implies less precise accounting though.

6 Summary

Algorithms like CHM, BPZ and CHD provide a fast, practical construction of Perfect Hash Functions. This makes it possible to use them in different fields from performance critical read-mostly data structures, like the routing tables, to size sensitive on-disk databases. NetBSD 6 is the first BSD to use them in the real world and more areas will be covered in the future.

Areas for open research and practical implementations outlined in this paper include finishing the implementation in the network stack and finding fast simple random hash functions to replace the remaining use cases of GNU gperf.

References

[1] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *In Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*, pages 139–150. Springer LNCS, 2007.

[2] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43:257–264, 1992.

[3] F. Botelho D. Belazzougui and M. Dietzfelbinger. Hash, displace, and compress. In *Algorithms – ESA*, pages 682–693, 2009.

[4] B. Jenkin. Hash functions. *Dr. Dobbs Journal*, September 1997.

[5] A. Klink and J. Wälde. Efficient denial of service attacks on web application platforms, 2011.

[6] S. Nilsson and G. Karlsson. IP-Address lookup using LC-tries, 1998.

[7] P. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, 33:677–680, June 1990.

[8] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. ACM SIGCOMM*, pages 25–35, 1997.