Near-Optimal Space Perfect Hashing Algorithms

Fabiano C. Botelho^{1,2} **and Nivio Ziviani**^{2}

[1]Department of Computer Science
Federal University of Minas Gerais, Belo Horizonte, Brazil

[2]Department of Computer Engineering
Federal Center for Technological Education of Minas Gerais, Belo Horizonte, Brazil

fabiano@decom.cefetmg.br, nivio@dcc.ufmg.br

Abstract. *A perfect hash function (PHF) is an injective function that maps keys from a set S to unique values. Since no collisions occur, each key can be retrieved from a hash table with a single probe. A minimal perfect hash function (MPHF) is a PHF with the smallest possible range, that is, the hash table size is exactly the number of keys in S. Differently from other hashing schemes, MPHFs completely avoid the problem of wasted space and wasted time to deal with collisions. The study of perfect hash functions started in the early 80s, when it was proved that the theoretic information lower bound to describe a minimal perfect hash function was approximately 1.44 bits per key. Although the proof indicates that it would be possible to build an algorithm capable of generating optimal functions, no one was able to obtain a practical algorithm that could be used in real applications. Thus, there was a gap between theory and practice. The main result of the thesis filled this gap, lowering the space complexity to represent MPHFs that are useful in practice from $O(n \log n)$ to $O(n)$ bits. This allows the use of perfect hashing in applications to which it was not considered a good option. This explicit construction of PHFs is something that the data structures and algorithms community has been looking for since the 1980s.*

1. Introduction

The need to access items based on the value of a key is ubiquitous in Computer Science. Some types of databases are updated only rarely, typically by periodic batch updates. This happens for most data warehousing applications (see [Seltzer 2005] for more examples and discussion). In applications where the key set is fixed for a long period of time the construction of a minimal perfect hash function can be done as part of the preprocessing phase. For example, On-Line Analytical Processing applications use extensive preprocessing of data to allow very fast evaluation of certain types of queries. More formally, given a *static* key set $S \subseteq U$ of size n from a key universe U of size u, where each key is associated with satellite data, the question we are interested in is: what are the data structures that provide the best tradeoff between space usage and lookup time?

Perfect hashing is a space-efficient way of creating compact representation for a static set S of n keys. For applications with successful searches[1] the representation of a key $x \in S$ is simply the value $h(x)$, where h is a perfect hash function (PHF) for the set S of values considered. The word "perfect" refers to the fact that the function will map

[1]A *successful search* happens when the queried key is found in the hash table and an *unsuccessful search* happens otherwise.

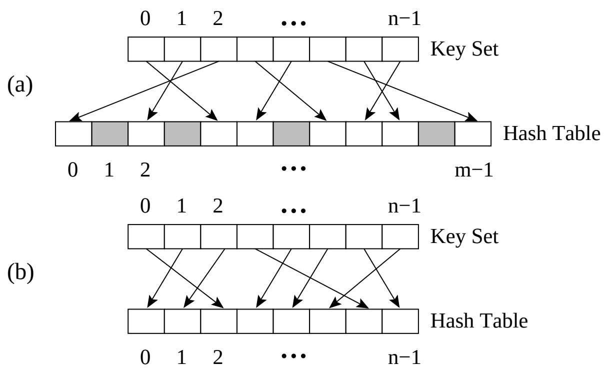

Figure 1. (a) Perfect hash function (b) Minimal perfect hash function.

the elements of S to unique values (is identity preserving). *Minimal perfect hash function* (MPHF) produces values that are integers in the range $[0, n-1]$, which is the smallest possible range. Figure 1(a) illustrates a PHF and Figure 1(b) illustrates an MPHF.

The study of perfect hash functions started in the early 80s, when it was proved that the theoretic information lower bound to describe a minimal perfect hash function was approximately 1.44 bits per key [Mehlhorn 1984]. Although the proof indicates that it would be possible to build an algorithm capable of generating optimal functions, no one was able to obtain a practical algorithm that could be used in real applications. Thus, there was a gap between theory and practice. The main result of the thesis filled this gap, lowering the space complexity to represent minimal perfect hash functions that are useful in practice from $O(n \log n)$ to $O(n)$ bits. This allows the use of perfect hashing in applications to which it was not considered a good option. This explicit construction of PHFs is something that the data structures and algorithms community has been looking for since the 1980s, as said by a reviewer of a prior submission: "*Taking into account the fact that people had been looking for such constructions all the time since the 1980s, this is a big achievement and might make the central result of the paper a candidate for...*".

The remainder of this paper is organized as follows. Section 2 discusses the main contributions. Section 3 discusses the impact of the results. Section 4 presents the conclusions. Section 5 discusses some ongoing work and future directions.

2. Key Contributions

The attractiveness of using PHFs and MPHFs depends on the following issues [Hagerup and Tholey 2001]: (i) the amount of CPU time required for generating the functions; (ii) the space requirements for generating the functions; (iii) the amount of CPU time required by the functions for each retrieval; and (iv) the space requirements of the description of the resulting functions to be used at retrieval time.

No previously known algorithm performs well for all these requirements. Usually, the space requirement for generating the functions is overlooked. That is why the algorithms in the literature cannot scale for key sets on the order of billions of keys. Also, as mentioned before, there is a gap between theory and practice on perfect hashing algorithms [Botelho 2008]. So, the main contributions of the thesis are:

1. We present a simple, practical and highly scalable perfect hashing algorithm that takes into account the four aforementioned requirements [Botelho et al. 2007, Botelho and Ziviani 2007, Botelho et al. 2009b]. When the input key set fits in

the internal memory available, it becomes an internal random Access memory algorithm, referred to as *RAM algorithm* from now on; otherwise, it becomes an external memory algorithm, referred to as *EM algorithm* from now on.

2. We provide a scalable parallel implementation of the EM algorithm, referred to as *parallel external memory* (PEM) algorithm from now on [Botelho et al. 2008a].
3. We present techniques that allow the generation of PHFs and MPHFs based on random graphs containing cycles [Botelho et al. 2005].
4. We show that the PHFs and MPHFs we have designed can now be used for applications in which they were not considered a good option in the past. In [Botelho et al. 2008b, Botelho et al. 2009a] we show that MPHFs provide the best tradeoff between space usage and lookup time when compared to other hashing schemes for indexing internal memory when static key sets are involved.
5. We have created the C Minimal Perfect Hashing Library [Botelho et al. 2006], referred to as *CMPH Library* from now on, that is a free software library available under the GNU Lesser General Public License (LGPL). The library was conceived for two reasons. First, we would like to make available our algorithms to test their applicability in practice. Second, we realized that there was a lack of similar libraries in the open source community.

We now describe the key contributions in the order they appear in the original thesis document [Botelho 2008]. For the sake of space, we do not provide extended details about each contribution. Please check the thesis document for details about the algorithms and implementations related to each contribution.

2.1. Random Access Memory and External Memory Algorithms

The RAM algorithm [Botelho et al. 2007, Botelho et al. 2009b] works on acyclic random graphs given by function values of uniform hash functions on the keys of S (see [Botelho 2008] for the definition of uniform hashing). The idea of basing perfect hashing on acyclic random graphs is not new, see e.g. [Majewski et al. 1996], but we proceed differently to achieve a space usage of $O(1)$ bits per key rather than $O(\log n)$ bits per key. We use r hash functions and acyclic hypergraphs with hyperedges $e(x) = \{h_0(x), \dots, h_{r-1}(x)\}$, for x a key, but add two tricks: (i) to key x assign an element $h_{i(x)}(x)$ of $e(x)$ such that the assignment $x \mapsto h_{i(x)}(x)$ is one-to-one on S; (ii) use a linear equation to calculate the index $i(x) \in [0, r-1]$ from x. This makes it possible to obtain a space usage of $c(r)\lceil\log(r+1)\rceil$ bits per key, for certain numbers $c(2), c(3) \dots$; the value that minimizes the cost per key is $r = 3$. The connection to acyclic random graphs allows us to perform a tight analysis and to optimize the space usage constant by using appropriate succinct data structures in a theoretically sound way.

The EM algorithm [Botelho and Ziviani 2007, Botelho et al. 2009b] is a result of a careful engineering process that uses a number of techniques from the literature to allow the generation of PHFs or MPHFs for sets on the order of billions of keys. The EM algorithm is the first step aiming to bridge the gap between theory and practice on perfect hashing. Therefore, it is the first algorithm that can be used in practice, has time and space usage carefully analyzed without unrealistic assumptions, and scales for billions of keys. We have designed two versions of the EM algorithm. The first one uses the hash functions described in [Botelho 2008], which guarantee that the EM algorithm can be made to work for every key set. The second one uses faster and more compact pseudo random hash functions proposed in [Jenkins 1997], referred to as heuristic EM algorithm, or simply

HEM algorithm from now on, because it is not guaranteed that it can be made to work for every key set. However, limited randomness often suffices in practice [Alon et al. 1999], and the HEM algorithm has worked for all key sets we have applied it to.

The RAM and EM algorithms generate in linear time PHFs and MPHFs that are evaluated in $O(1)$ time. The space requirements to describe the resulting functions depend on the relation between m and n. For $m = n$, the space usage is approximately $2.62n$ for the RAM algorithm and approximately $3.3n$ bits for the EM algorithm. For $m = 1.23n$, the space usage is approximately $1.95n$ bits for the RAM algorithm and approximately $2.7n$ bits for the EM algorithm. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n.

The main practical perfect hashing algorithms we found in the literature to compare the RAM, EM and HEM algorithms with are: Botelho, Kohayakawa and Ziviani [Botelho et al. 2005] (referred to as BKZ), Fox, Chen and Heath [Fox et al. 1992] (referred to as FCH), Majewski, Wormald, Havas and Czech [Majewski et al. 1996] (referred to as MWHC), and Pagh [Pagh 1999] (referred to as PAGH). For the MWHC algorithm we used the version based on random hypergraphs with $r = 3$. We did not consider the one that uses random graphs with $r = 2$ because it is shown in [Botelho et al. 2005] that the BKZ algorithm outperforms it.

Table 1 shows that the RAM (for $r = 3$), EM and HEM algorithms are the fastest ones to generate the functions and the resulting functions are the most compact. The performance of both EM and HEM algorithms is quite surprising once they use external memory at generation time and the other algorithms do not. However, as the key set is stored in external memory, all the other algorithms scan the whole key set everytime a failure occurs, whereas both EM and HEM algorithms simply scan the whole key set once and maps it to a set of fixed length fingerprints. Also, the whole key set is broken into buckets with at most 256 keys and the memory is accessed in a less random fashion, implying in fewer cache misses.

Table 1. Comparison of the algorithms for constructing MPHFs considering generation time and storage space, and using $n = 3,541,615$ for the two collections.

Algorithms		Generation Time (sec)		Storage Space	
		4-byte Integers	URLs	Bits/Key	Size (MB)
RAM	$r = 2$	11.39 ± 1.33	16.73 ± 1.89	3.60	1.52
	$r = 3$	5.46 ± 0.01	6.74 ± 0.01	2.62	1.11
EM		5.86 ± 0.17	7.68 ± 0.22	3.31	1.40
Heuristic EM		5.56 ± 0.16	6.27 ± 0.11	3.17	1.34
BKZ		9.22 ± 0.63	11.33 ± 0.70	21.76	9.19
FCH		$2,052.7 \pm 530.96$	$2,400.1 \pm 711.60$	4.22	1.78
MWHC		5.98 ± 0.01	7.18 ± 0.01	26.76	11.30
PAGH		39.18 ± 2.36	42.84 ± 2.42	44.16	18.65

Figure 2 illustrates that the both versions of the EM algorithm is able to generate an MPHF for a key set of 1.024 billion keys in less than 46 minutes, using a commodity PC. There is no algorithm in the perfect hashing literature that gets even close.

2.2. Parallel External Memory Algorithm

The Parallel External Memory (PEM) algorithm [Botelho et al. 2008a] allows to distribute the construction, description and evaluation of the resulting functions, which is

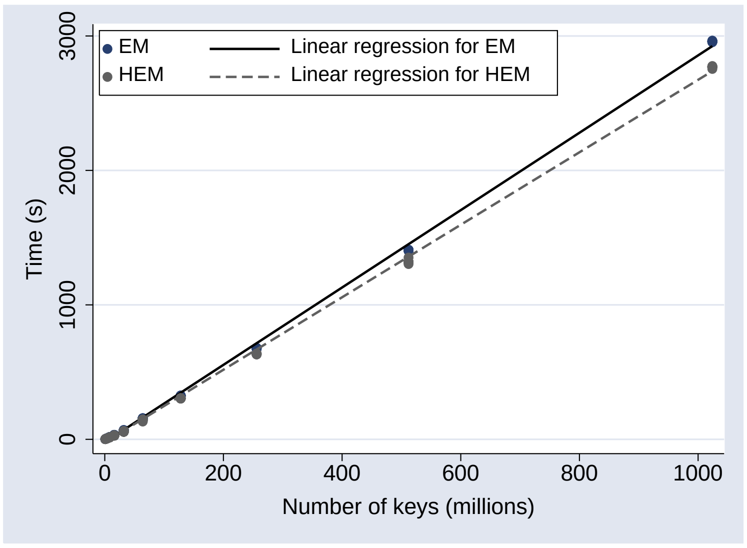

Figure 2. Number of keys in S versus generation time for the EM algorithm and the heuristic HEM algorithm. The solid line corresponds to a linear regression model for the generation time.

of fundamental importance when the key set size increases considerably. For instance, using a 14-computer cluster the PEM algorithm generates an MPHF for 1.024 billion URLs in approximately 4 minutes, achieving an almost linear speedup. Also, for 14.336 billion 16-byte random integers evenly distributed among the 14 participating machines the PEM algorithm outputs an MPHF in approximately 50 minutes, resulting in a performance degradation of 20%. To the best of our knowledge there is no previous result in the perfect hashing literature that can be implemented in a parallel way to obtain better scalability and performance than the results presented by the PEM algorithm.

2.3. MPHFs and Random Graphs with Cycles

The reason to use random graphs with cycles comes from the fact that the functions are generated faster and are more compact than the ones generated based on acyclic random graphs. This is because both the generation time and the space usage of the resulting functions depend on the number of vertices in the random graphs and the acyclic ones are more sparse. That is, the ratio between the number of vertices and number of edges must be larger than two.

Our result presented in [Botelho et al. 2005] improved the space requirement of one instance of the algorithms proposed in [Majewski et al. 1996]. Both algorithms are linear on n, but our algorithm runs 59% faster than the one in [Majewski et al. 1996], and the resulting MPHFs are stored using half of the space. The resulting MPHFs still need $O(n \log n)$ bits to be stored. As in [Majewski et al. 1996], the algorithm assumes uniform hashing and needs $O(n)$ computer words of the Word RAM model to construct the functions. Recently, using ideas similar to the ones presented in [Botelho et al. 2005], we have optimized the version of the RAM algorithm that works on random bipartite graphs to output the resulting functions 40% faster when cycles are allowed. These results are presented in [Botelho 2008, Chapter 6] and are being prepared for publication.

2.4. Indexing Internal Memory with MPHFs

We have shown that MPHFs provide the best tradeoff between space usage and lookup time when compared to other hashing schemes for indexing static key sets in internal memory [Botelho et al. 2008b]. It was not the case in the past because the space overhead to store MPHFs was $O(\log n)$ bits per key for practical algorithms [Majewski et al. 1996]. However, the MPHFs generated with the RAM algorithm [Botelho et al. 2007, Botelho et al. 2009b] require approximately 2.6 bits per key of space to describe the function and can be evaluated in $O(1)$ time, and completely changed that scenario. In [Botelho et al. 2009a] we extended our prior study in two aspects. First, we have designed an optimization of the MPHFs that considerably improves their lookup time performance. Second, we have surveyed the main hashing schemes available in the literature and added four other methods to our comparative study.

We have shown that other hashing schemes cannot outperform minimal perfect hashing considering lookup time even when the hash table occupancy is lower than 20%. An MPHF requiring just 2.6 bits per key of storage space is able to store sets in the order of 10 million keys in a 4 MB CPU cache, which is enough for a large range of applications. Besides, the space overhead of minimal perfect hashing is within a factor of $O(\log n)$ bits lower than other hashing schemes.

2.5. CMPH Library

The CMPH Library [Botelho et al. 2006] contains a professional implementation of our main results and is the state-of-the-art software for perfect hashing. We have received very good feedbacks about the practicality of the library. For instance, it has received more than $3,300$ downloads (July 2009) and is incorporated by two Linux distributions: Debian and Ubuntu This have contributed to make the results of this thesis becoming widely used in a short period of time, which usually takes much more time.

3. Impact of the Results

Three published papers have 21 citations (excluding self-citations) and one of them has more than 145 *downloads* in the ACM Portal in the last 12 months. Two papers that cite our results mention that we have the first really practical perfect hashing result in 20 years of research [Edelkamp and Sulewski 2008]. As mentioned before, the CMPH library has more than $3,300$ downloads up to July 2009 and is incorporated by two Linux distributions: Debian and Ubuntu, and has been used for applications that were inviable in the past. For instance, the results are being used into the products of two big companies hosted in California, United States: (i) Symantec Incorporation, and (ii) Data Domain Incorporation. Due to the impact of the results in the products of Data Domain Inc. (company with a net revenue exceeding 270 million dollars in 2008 and an expected growth of 100% in 2009), Fabiano C. Botelho was offered a position and will join the team of the company from August 2009 on. Besides, some of the knowledge acquired in the doctorate process was used in a book [Ziviani and Botelho 2006] that has sold more than $1,500$ copies.

4. Conclusions

In the thesis here summarized we have designed a time efficient, highly scalable and near-optimal space perfect hashing algorithm. In a 64-bit architecture our algorithm is able to deal with key sets of size up to $n = 1.8 \times 10^{21}$. The resulting functions are evaluated in $O(1)$ time. The space necessary to describe the functions takes a constant number of bits per key, and it is within a factor of two from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs, something that has not been achieved by previous algorithms, except asymptotically for very large n. The algorithm is theoretically well understood and is the first one with theoretical properties that scale for billions of keys and can be used in practice. The algorithm is suitable for a distributed and parallel implementation, as the one presented in [Botelho et al. 2008a], which is able to generate an MPHF for a set of 14.336 billion 16-byte integer keys in 50 minutes using 14 commodity PCs, achieving an almost linear speedup. We have shown that MPHFs provide the best tradeoff between space usage and lookup time when compared to other hashing for indexing internal memory when static key sets are involved.

5. Ongoing and Future Work

We strongly believe that our results on perfect hashing and the advent of solid state disks, which are built based on flash memory technology, have a perfect match to improve the performance of computer systems in several contexts. For example, this has been successfully done in [Edelkamp and Sulewski 2008]. So, we are working on very promising applications in the Information Retrieval field. Besides, we are working on three more papers to be submitted in 2009. The first paper is a joint work with Professor Nicholas C. Wormald from the Department of Combinatorics and Optimization at University of Waterloo, the second one is a journal paper that extends the results presented in [Botelho et al. 2008a], and the third one is a journal paper that extends the results presented in [Botelho et al. 2005].

Acknowledgements

We thank the partial support given by the Brazilian National Institute of Science and Technology for the Web (grant MCT/CNPq 573871/2008-6), Project InfoWeb (grant MCT/CNPq/CT-INFO 550874/2007-0) and CNPq Grant 30.5237/02-0 (Nivio Ziviani).

References

Alon, N., Dietzfelbinger, M., Miltersen, P. B., Petrank, E., and Tardos, G. (1999). Linear hash functions. *Journal of the ACM*, 46(5):667–683.

Botelho, F. C. (2008). *Near-Optimal Space Perfect Hashing Algorithms*. PhD thesis, Federal University of Minas Gerais. Supervised by Nivio Ziviani, `http://www.decom.cefetmg.br/docentes/fabiano_botelho/en/publications.html`.

Botelho, F. C., Galinkin, D., Meira-Jr., W., and Ziviani, N. (2008a). Distributed perfect hashing for very large key sets. In *Proceedings of the 3rd International ICST Conference on Scalable Information Systems (InfoScale'08)*. ACM Press. One non self-citations from scientific articles in Google Scholar.

Botelho, F. C., Kohayakawa, Y., and Ziviani, N. (2005). A practical minimal perfect hashing method. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 488–500. Springer LNCS vol. 3503. Seven non self-citations from scientific articles in Google Scholar.

Botelho, F. C., Lacerda, A., Menezes, G. V., and Ziviani, N. (2009a). Minimal perfect hashing: A competitive method for indexing internal memory. *Information Sciences*. Submitted.

Botelho, F. C., Langbehn, H. R., Menezes, G. V., and Ziviani, N. (2008b). Indexing internal memory with minimal perfect hash functions. In *Proceedings of the 23rd Brazilian Symposium on Database (SBBD'08)*, pages 16–30.

Botelho, F. C., Pagh, R., and Ziviani, N. (2007). Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADs'07)*, pages 139–150. Springer LNCS vol. 4619. Ten non self-citations from scientific articles in Google Scholar.

Botelho, F. C., Pagh, R., and Ziviani, N. (2009b). A scalable and near-optimal space perfect hashing algorithm. *Transactions on Algorithms*. Submitted.

Botelho, F. C., Reis, D., and Ziviani, N. (2006). CMPH: C minimal perfect hashing library. Free Software Library. More than three thousands downloads by February 2009.

Botelho, F. C. and Ziviani, N. (2007). External perfect hashing for very large key sets. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM'07)*, pages 653–662. ACM Press. Four non self-citations from scientific articles in Google Scholar.

Edelkamp, S. and Sulewski, D. (2008). Flash-efficient ltl model checking with minimal counterexamples. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM'08)*, pages 73–82, Washington, DC, USA. IEEE Computer Society.

Fox, E. A., Chen, Q. F., and Heath, L. S. (1992). A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'92)*, pages 266–273.

Hagerup, T. and Tholey, T. (2001). Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, pages 317–326. Springer LNCS vol. 2010.

Jenkins, B. (1997). Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9).

Majewski, B. S., Wormald, N. C., Havas, G., and Czech, Z. J. (1996). A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554.

Mehlhorn, K. (1984). *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag.

Pagh, R. (1999). Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures (WADS'99)*, pages 49–54.

Seltzer, M. (2005). Beyond relational databases. *ACM Queue*, 3(3).

Ziviani, N. and Botelho, F. C. (2006). *Projeto de Algoritmos com Implementações em Java e C++*. Thomson Learning.