Blooming Trees for Minimal Perfect Hashing

Gianni Antichi, Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci
Dept. of Information Engineering, University of Pisa, ITALY
Email: {gianni.antichi, domenico.ficara, stefano.giordano, gregorio.procissi, fabio.vitucci}@iet.unipi.it

***Abstract*—Hash tables are used in many networking applications, such as lookup and packet classification. But the issue of collisions resolution makes their use slow and not suitable for fast operations. Therefore, perfect hash functions have been introduced to make the hashing mechanism more efficient. In particular, a minimal perfect hash function is a function that maps a set of n keys into a set of n integer numbers without collisions. In literature, there are many schemes to construct a minimal perfect hash function, either based on mathematical properties of polynomials or on graph theory. This paper proposes a new scheme which shows remarkable results in terms of space consumption and processing speed. It is based on an alternative to Bloom Filters and requires about 4 bits per key and 12.8 seconds to construct a MPHF with 3.8×10^9 elements.**

I. Introduction

Hash tables are frequently used in networking applications. They can be found in compilers, language translation systems, and information retrieval. But the issue of collisions resolution makes their use slow and not proper for fast operations. Therefore, perfect hash functions have been introduced to alleviate these limitations and to improve performance. A perfect hash function maps a static set of n keys into a set of m integers without collisions, where m is greater than or equal to n. If m is equal to n, the function is called minimal.

Minimal perfect hash functions (MPHFs) are widely used for memory efficient storage and fast retrieval of items. They can be used also for security purposes: the capability of efficiently revealing the presence of certain strings allows for a fast detection of attacks or for determining which data have to be anonymized in privacy-preserving approaches [1]. Given the high operating speeds of current links, item retrieval must be very fast. Moreover, item sets can be very large (e.g., search engines are nowadays indexing tens of billions of pages), thus these algorithms must be very space-efficient.

To summarize, the goodness of a MPHF scheme depends on the time and the space needed for its construction, on the time required by the MPHF for each retrieval of an element and on the number of bits needed to represent the element.

While CAM/TCAM hardware is a fast (yet energy-greedy) alternative to implement the same functions, the recent general trend for energy savings and the need for cheap and general implementation contribute to keep MPHFs important and attractive in network devices.

This work presents a new technique to construct a minimal perfect hash function by using specific data structures based on Blooming Trees [2]. The main objectives are 1) simple construction process, 2) fast retrievals, and 3) memory savings. The target platforms are network processors (NPs) or general purposes processors (GPPs) that provide the "popcount" instruction to compute the number of "1" bits in a word (for instance Intel® Itanium [3], the future Nehalem [4] and the IXP2800 [5], AMD® Phenom [6] and IBM® Power6 [7]).

The next section introduces the major results about MPHF construction. Sec. III illustrates the data structures that will be used. Sec. IV describes the overall MPHF algorithm, while sec. V computes its parameters and complexity. Finally, sec. VI shows simulation results and the comparison with the other most efficient solutions proposed in the literature.

This work has been partially supported by the European Project FP7-ICT PRISM, contract number 215350.

II. Related Works

In this section, we present the major results about the construction of MPHF. For further details, [8] gives a comprehensive survey on perfect hashing. Fredman, Komlós and Szemerédi [9] present a space efficient structure to construct MPHFs, which uses a space of the order of $n + o(n)$. The construction time of this model, based on hashing properties, is $O(n)$, and the same result is also obtained in [10], [11].

Mehlhorn [12] shows that at least 1.44 bits per key are needed to represent a MPHF. Fox et. al. [13] illustrate an algorithm whose encoding size is very close to such a theoretical bound (i.e., around 2.5 bits per key) and which uses the well-known *mapping-ordering-searching* scheme. However, [8] proves that such a scheme has exponential running times.

Pagh [11] proposes a new way of constructing MPHFs through randomized algorithms. The form of the resulting function is $h(x) = f(x) + d[g(x)]mod(n)$, where f and g are hash functions and d is a set of values to resolve collisions. The hash function description occupies $O(n)$ words and can be constructed in $O(n)$ expected time.

Czech et al. [10] introduce a new algorithm for MPHF which preserves the elements order. It involves the generation of random graphs; the time complexity is $O(n)$ and the space required to store the function is $O(n \log n)$, which is optimal for order preserving MPHFs [8]. This algorithm takes $32.9s$ to construct a MPHF for 524288 keys on a Sequent machine.

Botelho et al. [14] propose a solution based on the classic divide and conquer technique, which is capable of generating MPHFs for sets of billion of keys. The construction time is $O(n \log n)$, the evaluation of $h(x)$ requires 3 memory accesses for any key x and the description of h takes a number of up to 9 bits for each key, which is optimal for huge sets.

978-1-4244-2324-8/08/$25.00 © 2008 IEEE.

To the best of our knowledge, the solution which offers the best tradeoff between construction time and storage space is illustrated in [15]. It uses r-uniform random hypergraphs given by function values of r hash functions on the keys to be processed. Such an algorithm will be the reference for the performance evaluation of our solution.

Finally, [16] introduces a novel scheme for MPHF which requires about 8.6 bits per key. The construction is several orders of magnitude faster than existing perfect hashing schemes based on mapping-partitioning-searching model, because searching is avoided. Bloom Filters (BFs) are employed, which are known for simplicity and speed. This schemes, running on a Pentium IV, needs $7.73s$ to construct a MPHF for 3.8 millions of keys and $125ms$ for 110 thousands of keys. It inspires this work in the use of BF-like structures for MPHFs. Instead of standard BFs, a composed data structure is used: a first level is given by a Huffman Spectral Bloom Filter (HSBF) [17] while the remaining part is based on a novel filter, the so–called Blooming Tree (BT) [2]. This way, a novel method is proposed, which allows for an easy MPHF construction and fast retrieval, with low memory requirements.

III. Blooming Tree

The idea of Blooming Trees [2] is constructing a binary tree upon each element of a plain BF, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes. The aim is to achieve both low false positive probability and low memory requirements, while the drawback is the increased cost in lookup operation. The latter can be mitigated by the low memory consumption, that enables the deployment of the structure in faster on-chip memories.

To build a Naive Blooming Tree (NBT) for n elements, $L+2$ layers are defined:

- a plain BF (B_0) with k_0 hash functions h_j ($j = 1 \dots k_0$) and m bins such that $m = nk_0/\ln 2$;
- L layers ($B_1 \dots B_L$), each composed of m_i ($i = 1 \dots L$) blocks of 2^b bits;
- a final layer (B_{L+1}) composed of c-bits counters.

The j-th hash function h_j provides a $\log_2 m + L \times b$ bit long output: the first group ($s_{0,j}$) of $\log_2 m$ bits is used to address the BF at layer 0, the other $L \times b$ bits are divided into L substrings ($s_{1,j} \dots s_{L,j}$) of b bits, one for each layer.

The lookup for an element σ consists of a check on k_0 elements in the BF (layer 0) and an exploration of the corresponding k_0 "branches" of the Blooming Tree. The overall process of lookup is accurately explained in [2].

An optimized version of BT [2] follows from some observations about NBTs. In particular, the "zero-blocks" are used to stop the "branch" from growing as soon as the absence of a collision is detected in a layer (which entails for construction the absence of collisions in all the upper layer of the branch). However, to keep construction and lookup possible, the Optimized BT (OBT) employs a bitmap and an array of hash substrings for each layer. The array of substrings for a certain layer is composed of all the hash substrings that complete the hash of the "branches" that stop at that layer, while the bitmap addresses such substring array.

IV. The MPHF construction

Our algorithm is based on the statement that, given an ordering algorithm g and a set S, a MPHF of an element $x \in S$ is simply the position of x in the given ordering scheme:

$$\text{MPHF}(x) = \underset{S,g}{\text{position}}(x) \tag{1}$$

A. Using the Naive Blooming Trees

The basic idea of our algorithm is to use a BT as ordering algorithm for the set of elements S we have:

$$\text{MPHF}(x) = \underset{S,BT}{\text{position}}(x) \tag{2}$$

In particular, the NBT can be used for this purpose with no modifications. The construction is exactly the same as the one described in section III, using a single hash function. All we need to care is to make sure that the counters at layer $L+1$ are all equal to 1, which means that all the elements have been separated. In this situation, in order to evaluate MPHF(x), a single lookup operation is required: once the corresponding (say, the j-th) counter of x in layer B_{L+1} is found, we return j (obtained through a simple popcount) as the result.

We need to design the structure to have very low probability of collisions in the last layer (B_{L+1}). This, in turn, results in a high probability of obtaining a successful MPHF construction in a few attempts. If the construction is successful, we achieve our ordering scheme: the element that falls into the first counter is hashed to 0, the element falling into the second one is hashed to 1, and so on. Therefore, our hash function is perfect and also minimal, in that we assign the first n integers as the results of hash retrieval for n elements.

B. Using the Optimized Blooming Tree and the HSBF

The OBT is an elaborated version of the NBT that improves memory efficiency, thus being attractive for our purposes. The idea is blocking branch from growing, as soon as an element does not experience any collision, by using the zero-blocks as leaves of the trees. However, this expedient removes the last layer, which till now provided a simple way to compute a MPHF by means of popcounts.

Recall that the problem lies in how to compute the number of elements at the left of a given element x. Our idea to solve this problem is to divide the procedure into two steps:

- find the tree which x belongs to (we shall call it T_x) and compute the number of elements at the T_x's left;
- compute the number of leaves at the left of x within T_x.

In order to do so, we propose the HSBF [17] as the first level of the BT, instead of the standard BF [2]. The HSBF is composed of a series of bins encoded by Huffman coding so that a value j translates into j "1s" and a trailing "0". Therefore, the first step (i.e.: computing the number of elements in the trees at T_x's left) is obtained by a popcount in the HSBF of all the bins at the left of x's bin. As for the second step, we have to

explore (from left to right) the tree T_x until we find x, thus obtaining its position within the tree. The sum of these two components gives the hash value to be assigned:

$$\text{MPHF}(x) = \text{popcount}(\text{HSBF}[x]) + \underset{T_x}{\text{position}}(x) \qquad (3)$$

Notice that, in a standard BT, the popcount in the first layer gives the block to be read at the next layer. To achieve the same functionality in an HSBF, it suffices to count the number of bins greater than 0 (i.e.: the number of "10" bit-sequences).

The HSBF is divided into B sections of D bins, which are addressed through a lookup table. Each entry of this table keeps all the necessary information for a section: the starting address in memory, the number of elements that fall in the previous sections (which are computed by means of popcounts, as stated above), and the number of "10" bit-sequences found in the previous sections. The OBT has a maximum of L layers ($B_1 \dots B_L$), each composed of blocks of 2^b bits.

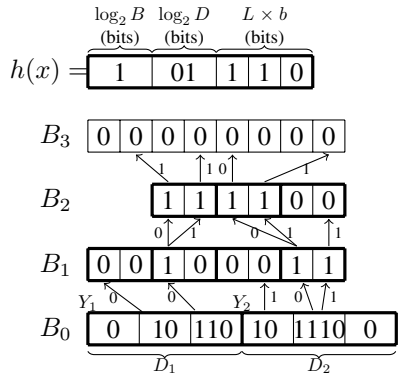
A hash function $h(\cdot)$ is used. Its output is $\log_2 B + \log_2 D + Lb$ long bits: the first $\log_2 B$ bits indicate the section and address the lookup table, the subsequent $\log_2 D$ bits index the bin within the section in the HSBF, while the last ones are divided into substrings of b bits, one for each BT layer.

A simple example (see fig.1) clarifies the procedure. Let us assume $B = 2$, $D = 3$, and $b = 1$: hence, the hash output is 6-bits long. Let us suppose $h(x) = 101110$. The first bit is used to address the lookup table: it points to the second entry. We read the starting address of section D_2 and that 3 elements are in the previous sections. Now we use the next two bits of $h(x)$ to address the proper bin in section D_2: "01" means the second bin. The popcount of the previous bins in the section indicates that another element is present (so far the total of elements at T_x's left is 4). Then we care about T_x: to move up to the next layer, we both use the third information in the table and count the number of "10"s in the previous bins of this section. The sum shows that, before our bin, 3 bins are not equal to 0, so we move to the fourth block in layer B_1.

Here, the fourth bit of $h(x)$ allows to select the bit to be processed: the second one. But we want to know all the T_x's leaves at x's left. Hence, we must explore all the branches starting from the first bit of the block and count the number of zero-blocks we find: it is 2 (now, the counter reads 6). Regarding the second bit, a popcount in layer B_1 indicates the third block in layer B_2: it is a zero-block, so we have found the block representing our element: x is the 7-th element in our ordering scheme. Then $\text{MPHF}(x) = 6$.

An obvious objection is that a lookup may require many jumps and be expensive since we need to explore, on average, half a tree to find our element. However, this is not a big issue because all the nodes of a tree at the same layer are contiguous in memory and can be accessed (and cached) in a single memory reference. Hence, the number of accesses for an element is simply the depth of the tree it belongs to.

Finally notice that, in the evaluation process, bitmaps and hash substrings tables have not been used; therefore, after the MPHF construction, they can be removed from memory.



Lookup Table

Start addr.	Prev. elements	Prev. "10"s
Y_1	0	0
Y_2	3	2

Figure 1. Example of hash retrieval by using OBT and HSBF.

C. Using a more efficient version

Another potential improvement becomes clear as we make the following remark on the above described structure: whenever a bin in the HSBF is equal to 1 (i.e.: it reads "10"), it is a waste of memory to allocate its zero-block at the next layer, because only a single element falls into it. Since the probability of having a bin equal to 1 is larger than the probability of large bin values in a CBF (see eq. (4) in the next section), this improvement notably reduces the average cost in terms of lookup time and memory size.

The construction process does not change but, after the construction, the structure can be reduced according to the previous discussion. Also, the lookup table must be modified: the third element of each entry must now indicates the number of "110" bit-sequences in the previous sections of the HSBF, because only bins strictly greater than 1 have a corresponding block at layer 1 under this new scheme.

The example in fig. 2 shows the reduction of the structure of fig. 1: we observe the deletion of the first and the third blocks in B_1, which were related to the second and the fourth bins in the HSBF (the "10" bins), and the change of the lookup table.

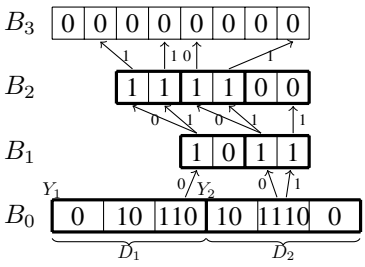
V. Complexity and properties

In order to simplify the rest of the analysis, it is useful to remind one of the main results of [2]:

$$P_i(\varphi) \simeq \frac{e^{-\alpha_i}\alpha_i^{\varphi}}{\varphi!} = \text{Poisson}(\alpha_i, \varphi) \quad \text{with } \alpha_i = 2^{-i}\ln 2 \qquad (4)$$

Eq. (4) claims that the number of elements φ colliding in any block of layer i can be well-approximated by a Poisson pmf with parameter α_i. This result provides a tool to design our MPHF. In particular, we can compute the number of layers needed to guarantee a fast construction ("fast", here, means "within one or few trials") of our BT by simply setting:

$$n \times P_{L+1}(2) \simeq q \qquad (5)$$



Lookup Table

Start addr.	Prev. elements	Prev. "10"s
Y_1	0	0
Y_2	3	1

Figure 2. Example of hash retrieval in the optimized structure.

where $q \leq 1$ is the probability of having at least a bin greater than 1 at layer $L+1$; in addition, q corresponds to the probability of the unlucky event that we need to retry the construction (it is also called the failure probability [16]).

In that event, a different approach can be pursued by just adding extra layers until the collisions disappear. This requires, of course, the output of the hash function $h(\cdot)$ to be wider than $\log_2 B + \log_2 D + Lb$ bits, but it can be less expensive than restarting the entire construction from scratch.

A. Memory requirements

The average size of our MPHF can be computed as the sum of its components. As for the OBT, since we do not need all its complementary structures such as bitmaps and hash substrings, we compute its size as the sum of leaves and non-leaves nodes. In the structure described in section IV-B the number of leaves is simply the number of elements n. However, in the optimized structure described in section IV-C, we delete the layer 1 leaves, thus obtaining $n - m_0 P_0(1) = n(1 - P_0(1)/\ln 2)$ leaves. On the other hand, the number of non-leaves nodes can be computed as $m_0 P_i(\varphi > 1)$. Thus the average memory size of our OBT is:

$$S_{OBT} = 2^b \left(n(1 - P_0(1)/\ln 2) + m_0 \sum_{i=1}^{N_l} P_i(\varphi > 1) \right) \quad (6)$$

where N_l is the number of required layers and $P_i(\varphi > 1) = 1 - P_i(0) - P_i(1)$ can be computed by means of (4).

When dealing with perfect hash functions, it is common to express the memory requirements in terms of bits per key:

$$S/n = 2^b \left(1/2 + \frac{1}{\ln 2} \sum_{i=1}^{N_l} P_i(\varphi > 1) \right) + 1 + \frac{1}{\ln 2} \quad (7)$$

A first comment is that $b = 1$ is the less expensive choice in terms of memory consumption. Larger values of b reduce the depth of the blooming tree (that is the number of non-leaves nodes) but its contribution to S/n in (7) is negligible. Therefore, $b = 1$ shall be the preferred setting hereafter. Moreover, we notice that the optimization process discussed in IV-C reduces the number of bits-per-key metric of 0.5 bits.

Table I
MEMORY REQUIREMENTS IN BITS/KEY.

b	W_{HSBF}	W_{OBT}	ΔS_{HSBF}	ΔS_{OBT}	S'	S_{tot}
1	512	512	0.45	0.10	3.63	4.18
1	1024	1024	0.23	0.06	3.63	3.92
1	2048	2048	0.11	0.03	3.63	3.77
2	512	512	0.45	0.10	4.82	5.37
2	1024	1024	0.23	0.06	4.82	5.11
2	2048	2048	0.11	0.03	4.82	4.96

However a number of tables are needed in order to make the lookup phase faster and more manageable. In fact, both the HSBF and the upper layers can be divided in sections so that accessing them or computing a popcount requires less time. We already discussed in section IV-B about the tables for the HSBF. Now, we also consider dividing each layer of the OBT into sections and adding, for each layer, a table whose j-th entry reports the popcount of all bits before section j.

Naturally, the increment in memory requirements introduced by these tables depends on B and D. If we focus on the bits/key metric, we can compute the table cost in memory through b and W only (i.e., the bit size of sections). Tab. I reports the consumption in bits/key for the total structure (S_{tot}) along with the contributions of lookup tables for the HSBF (ΔS_{HSBF}) and the OBT (ΔS_{OBT}), which are added to the main structure S'. A good choice is $W = 1024$ bits: all the tables cost only 0.29 bits per key, thus bringing the final memory requirement to 3.92 bits per key. Moreover, in modern 64-bits processors, 1024 bits represent 16 words only, and can be read in a single memory access. However, other values of W do not significantly change the final memory budget.

B. Hash evaluation cost

In the following study on the average cost of a lookup, we focus on the number of memory accesses. Indeed a memory access commonly requires hundreds of clock cycles, thus accounting for almost the totality of the hash evaluation cost.

During a lookup, we have to compute a hash function and use its output to address the lookup table and the HSBF. Moreover, if the bin we read reports a collision (i.e.: more than 1 element falls into it), we need to explore a certain number of layers according to the depth of the resulting tree. Eq. (4) comes in handy also in this computation. It expresses the pmf of m bins, but we do not care about empty bins. Therefore we need to normalize the pmf in (4) by dividing it by $(1 - P_0(0)) = 1/2$:

$$P'_i(\varphi) = \begin{cases} \frac{P_i(\varphi)}{1-P_0(0)} = 2 \times P_i(\varphi) & \varphi \geq 1 \\ 0 & \varphi = 0 \end{cases} \quad (8)$$

Of course, $P'_i(0) = 0$ because we will not lookup empty bins. Then $n \times P'_0(1)$ times we will access only the lookup table and the HSBF. This costs two memory accesses (if a HSBF section is read in a single access). In all other cases ($n \times (1 - P'_0(1))$ times), we have a tree to explore.

As previously mentioned, in our construction all nodes of a tree at the same layer are contiguous in memory. This means

Table II
ALGORITHM COMPARISON.

Algorithm		Evaluation		Construction Time		bits/key
		time(s)	mem.ref.	mean(s)	std.dev	
Our	$m = 2^{22}$	1.21	3.1	12.78	0.11	4.02
	$m = 2^{23}$	0.98	2.53	13.37	0.14	4.75
BPZ		1.35	-	18.37	4.41	3.60
BL		-	2.38	7.73	-	9.1

that, as a matter of fact, when accessing a given node we read (and cache) all the other nodes at the same layer. Therefore the average number of memory accesses for the tree exploration is simply the average tree depth $\overline{d} = \sum_{i=1}^{L+1} i \times P'_i(1) \simeq 2.4$. Finally, the average number of memory accesses is:

$$\overline{w} = 2 + (1 - P'_0(1))\overline{d} \simeq 2.73 \quad (9)$$

It does not depend in any way on the number of elements, hence the lookup cost is $O(1)$.

VI. EXPERIMENTAL RESULTS

We compare our algorithm to the one proposed by Bonomi and Lu [16] (hereafter called BL) and to the fastest and least memory-requiring algorithm that we found [15] (BPZ). We are aware that, because of the processor evolution and the limited availability of the code of other algorithms, it is always difficult to present fair comparisons for algorithms. We tested an implementation of our algorithm (developed in C) on an Intel 2.4 Ghz Pentium 4 Core 2 Duo processor (4 MB L2-Cache), equipped with 4 GB of RAM and running Linux OS 2.6, while BPZ employed a 3.2 Ghz XEON (2 MB L2-Cache), with 1 GB of RAM running Linux 2.6 and BL describes its test platform simply as Pentium 4. Even if the processor we used is dual-core, this does not give us any advantage because our implementation is sequential and runs on a single core. This means also that, even if the L2-cache is 4 MB, only half of it is available (on average) to our algorithm. Both papers (on BPZ and BL) present their results for a similar number of keys ($3.541.615$ for BPZ and 3.8 million for BL) thus we set $n = 3.8 \times 10^6$ in our algorithm.

In tab. II we show a comparison of BPZ, BL and our algorithm in terms of construction and lookup time, as well as memory requirements. The lookup time is obtained by querying the MPHF for all the 3.8×10^6 keys in random order.

Since in our algorithm we would like to have $m \simeq n/\ln 2$, we have two choices for the first layer: $m = 2^{22}$ or $m = 2^{23}$. We tested both. In the first case, we measured 3.1 memory accesses (on average) and 4.02 bits per key, while in the second case we had a faster lookup, but an increment of about 0.7 bits per key in the size and of more than 0.5s in the construction time. Such values confirm the theoretical results of the previous section. As for the failure probability, by limiting the number of layers to 10, it turned out to be less than 5×10^{-4} in all cases. However, on processors such as the ones cited in sec. I, both construction and lookup times will decrease because of the high frequency of popcount calls.

Results clearly show an improvement in terms of construction and lookup time as compared to BPZ, at the cost of a slight increase in memory requirement. Instead, compared to BL, our solution halves the bits-per-key metrics, while slowing down lookup and construction processes.

VII. CONCLUSION

The paper presents a new solution for the construction of a minimal perfect hash function, which is useful in many networking applications. We use two novel data structures which we previously introduced in recent works: Huffman Spectral Bloom Filters and Blooming Trees. The overall multilayered structure is based on randomized algorithms and allows for remarkable memory savings. We compared our algorithm to the one proposed in [16], which uses another BF-like structure, and to the one described in [15] which, to the best of our knowledge, offers the best performance in terms of lookup time and memory consumption. Our solution shows about the same performance compared to BPZ and a remarkable memory saving with respect to BL. In details, it presents a consumption of about 4 bits/key and a retrieval time of about 1 second. Moreover, its specific design allows to take advantage of the new features of recent and future processors, both for networking and general purposes applications.

REFERENCES

[1] G. Bianchi, S. Teofili, M. Pomposini, "New directions in privacy-preserving anomaly detection for network traffic," in *Proc. of Network Data Anonymisation (NDA 2008)*, Virginia, USA, October 2008.

[2] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Blooming trees: Space efficient structures for data representation," in *Proc. of ICC'08*, Beijing, China, May 2008.

[3] *http://www.intel.com/design/itanium/documentation.htm.*

[4] *http://softwarecommunity.intel.com/isn/Downloads/Intel SSE4 Programming Reference.pdf.*

[5] *http://www.intel.com/design/network/products/npfamily/ixp2805.htm.*

[6] *http://vincent.amd.com/us-en/assets/content_type/white papers and tech docs/40546.pdf.*

[7] *http://www.power.org/resources/reading/PowerISA_V2.05.pdf.*

[8] Z. J. Czech, G. Havas, and B. S. Majewski, "Fundamental study - perfect hashing," *Theoretical Computer Science*, vol. 182, no. 1, August 1997.

[9] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with 0(1) worst case access time," *J. ACM*, vol. 31, no. 3, 1984.

[10] Z. J. Czech, G. Havas, and B. S. Majewski, "An optimal algorithm for generating minimal perfect hash functions," *Information Processing Letters*, vol. 43, no. 5, 1992.

[11] R. Pagh, "Hash and displace: Efficient evaluation of minimal perfect hash functions," in *Workshop on Algorithms and Data Structures*, 1999.

[12] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1984, vol. 1.

[13] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, "Practical minimal perfect hash functions for large databases," *Commun. ACM*, vol. 35, no. 1, 1992.

[14] F. C. Botelho, Y. Kohayakawa, and N. Ziviani, "An approach for minimal perfect hash functions for very large databases," Universidade Federal de Minas Gerais, Belo Horizonte, Brazil, Tech. Rep., 2006.

[15] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *Springer-Verlag Lecture Notes in Computer Science*, vol. 4619, 2007.

[16] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *Proceedings of International Symposium on Information Theory 2006*, 2006.

[17] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting bloom filters," in *Proc. of INFOCOM '08*, Phoenix, AZ, USA, April 2008.