International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 4 Issue: 6 619 – 623

A Survey of Hashing Techniques for High Performance Computing

G M Sridevi
Asst. Professor, Dept. of ISE
SJBIT, Bengaluru, India
e-mail: sridevi.gereen87@gmail.com

M V Ramakrishna
Professor, Dept. of ISE
SJBIT, Bengaluru, India
e-mail: mvrama@yahoo.com

***Abstract*—** Hashing is a well-known and widely used technique for providing O(1) access to large files on secondary storage and tables in memory. Hashing techniques were introduced in the early 60s. The term hash function historically is used to denote a function that compresses a string of arbitrary input to a string of fixed length. Hashing finds applications in other fields such as fuzzy matching, error checking, authentication, cryptography, and networking. Hashing techniques have found application to provide faster access in routing tables, with the increase in the size of the routing tables. More recently, hashing has found applications in transactional memory in hardware. Motivated by these newly emerged applications of hashing, in this paper we present a survey of hashing techniques starting from traditional hashing methods with greater emphasis on the recent developments. We provide a brief explanation on hardware hashing and a brief introduction to transactional memory.

Keywords- *Bloom Filters, Transactional Memory, Universal Hash Functions, Hardware Hashing*

*****

I. INTRODUCTION

Traditional hashing techniques used hash tables to provide constant access to files. Dynamic hashing techniques were developed in late 70s to deal with dynamic files that keep changing in size: extendible hashing [8], linear hashing [16] and dynamic hashing [11]. Much of the research during 60s and 70s dealt with overflow handling techniques and perfect Hashing for large files was developed in 80s [22]. Not much work was done on the implementation of the different methods in real time applications. Sprugnoli was the first to envision perfect hashing with an ideal goal of 1 access retrieval to files. Perfect Hashing can be achieved when the key set is of fixed size and known in advance [28]. Ramakrishna and Portice proposed a simple trial-and-error method to achieve perfect hashing for Hardware applications [25]. Perfect Hashing is now being used for networking applications. Hashing also finds multiple applications in hardware like translation look aside buffer, transactional memory systems, networking hardware [30, 13, 27]. Hash functions are used to implement Bloom filters. A Bloom filter is a data structure that is used to check if a given element is a in a set or not. Bloom Filters are currently being used in Google File System, Big Table, HBase, etc. A more recent application is in transactional memory which makes use of hardware hash functions, to keep track of read/write sets to detect conflicts between different transactions [13], [17].

Our aim is to survey the area with emphasis on modern developments. Hashing has come a long way since early days when 'hashing' was not considered a decent print word [10]. In the following section, we provide a brief description of different dynamic hashing schemes. This will be followed by a description of Hashing in Hardware. We describe the concept of universal hashing and a particular class of functions called H3. We give a brief explanation about the use of hashing in transactional memory followed by Perfect Hashing. Lastly we present the use of hashing in the implementation of bloom filters which has found its applications in various fields.

II. TRADITIONAL HASHING SCHEMES

In the early days, the main issue was how to handle overflows while storing data in memory. Many hashing techniques were introduced to address the issue. Linear probing or progressive overflow was proposed as a solution. This method looked for a free slot in consecutive memory spaces to store the overflow record. This method had some drawbacks where the average search length increased rapidly with the increase of packing density. In double hashing, when collision occurs, a second hash function was applied to the key to get a number c which was added to the original address to obtain the overflow address. The drawback of this method was that it moved the keys far away from their home address. Chained progressive overflow used pointers to link the keys having same home address together. Two pass loading was needed to ensure that a home address was occupied by a home record. To avoid overflow records from occupying home addresses, a separate overflow area was used to store the overflow records. To provide the advantage of simple indexing, scatter tables were used which had pointers to records and acted as an index. Research was done to improve the average access performance. Many papers were published with different techniques including repositioning the overflow records. Such methods include Robin Hood hashing in which the overflow record which probed to a longer distance was stored in the address space and the one with shorter probe distance was made to probe for the next free slot available [24]. These techniques were used both for main memory tables and large secondary storage which typically has b > 1 number of records/address.

III. DYNAMIC HASHING SCHEMES

In order to reduce the access time to files from secondary storage devices, many techniques have evolved over time such as indexing, B-Trees, B+ Trees and hashing. Among those, hashing is the most efficient file organization technique that provides O(1) access to files stored on secondary storage. While Hashing provides efficient access for unchanging files, Dynamic Hashing is used for files that grow and shrink dynamically.

A. *Extendible Hashing*

Extendible hashing is a technique that increases or decreases the number of slots in a hash table in proportion to the number of elements in the table. The directory size needs to be a power of 2. Hash addresses are obtained by a hashing function that gives a sequence of bits. Overflows are handled by doubling the directory which logically doubles the number of buckets. The overflow bucket is split. Extendible hashing allows insertions and deletions to occur without resulting in poor performance after many such operations. Fagin, et al states that extendible hashing can be used for reducing the number of page faults for accessing any data from dynamic files [8]. They analyzed and simulated the performance of extendible hashing and conclude that it is an attractive alternative to other access methods such as balanced trees and radix search trees.

B. *Linear Hashing*

Linear hashing is a dynamic hashing technique for disk-based files. The file grows or shrinks, one bucket at a time. Like Extendible hashing, it uses more bits of the hashed key as the address space grows but unlike Extendible Hashing, it does not make use of an explicit directory. A chain of overflow pages are maintained at the overflow bucket to handle the overflow. The address space is extended linearly, one bucket at a time. Litwin made a comparative study of the performance of various file access techniques like classical hashing, trees and virtual hashing and concludes that linear hashing provides best performance in comparison [16]. Larson proposed a simple method based on linear probing for handling overflow records in connection with linear hashing. He also presented a new method offering one-access retrieval for large dynamic files and necessary address computation, insertion and expansion algorithms are simulated [12].

IV. PERFECT HASHING

Perfect Hashing refers to hashing without overflows/collisions. Perfect hash functions are used for efficient storage of data in memory and fast access of items from static sets like words in human languages, reserved words in programming languages and routing tables. Sprugnoli used Direct Perfect hashing method to deal with small static sets [28]. He presented a hash function of the form $h(x) = (x + c)/N$ where c and N are constants which were determined by Quotient Reduction Method. For keys with non-uniform distribution, he proposed the use of hash functions of the form $h(x) = ((xq + d) mod\ m)/N$ to achieve better performance. He described a method called Remainder Reduction Method to evaluate the constants q, d, m and N. Direct Perfect Hashing is not suitable for larger sets. Fredman et. al used a scheme that implemented Sprugnoli's idea of segmentation for handling larger sets [33]. Du, Hsieh et. al proposed the use of Composite perfect hashing scheme in which the hash function was a composite of a set of hash functions $\{h_1, h_2, \ldots\ldots, h_s\}$ [32].Perfect Hashing has been investigated as a technique for large file organizations and for hardware applications like associative memory implementation [23, 26]. Most recently, different Perfect Hashing Techniques have received patents from US patent office for their application in different areas. Perfect hashing is used for faster pattern search, memory efficient storage, and faster access to static sets and so on. Perfect hashing can be used when the data set is known beforehand and it is possible to hash the data without collisions. Zhou et.al have proposed a IPV6 lookup approach based on hierarchical perfect hashing to provide faster access to routing tables [32]. Hierarchical perfect hashing was used to provide faster IP lookup. The authors have conducted experiments and concluded that perfect hashing improves the performance of the search even for large routing tables. Botelho et. al received a US patent for memory efficient cleansing of a de-duplicated storage system with the use of a perfect hash function [2]. They proposed a method to remove sensitive information from the memory to avoid misuse of the data. They presented a comparison of the memory requirement using different techniques like Reference counts, Bloom Filter, Perfect Hash and Bit Vector.

V. HARDWARE HASHING

Hashing is fundamental for achieving a high performance in computer architecture. In 1967, IBM hardware used hash tables for page address translation using bit extraction. Hashing is used extensively in hardware applications, such as page tables, for address translation. Hashing is also applied in bloom filters, Transactional Memory and networking applications. Different versions of hash functions like SHA (Secure Hash Algorithm) and MD(Message Digest) may be used to implement bloom filters. Bloom Filters may be used in different applications such as weak password dictionary, SPIE (Source Path Isolation Engine) Trace-back, cache sharing, networking and also in Transactional Memory [19, 3].

A. *Transactional Memory*

Multicore processors were introduced in the early 2000s and have become a necessity to handle the increasing workload. Now most systems are multi-core, such as, Intel Xeon E7-2850 which is a ten core processor. Multicore systems have two or more processing units called cores integrated onto a single chip package. The basic idea behind the use of multiple cores is to divide the workload among the cores that process the instructions in parallel thereby decreasing the overall computation time. The cores are interconnected to one another via different network topologies like bus, ring, mesh and crossbar. Multiple cores may share the memory with independent cache memory for each core. Data from main memory is fetched and stored in the cache. One of the issues faced in implementing a multicore system is to maintain data consistency between the shared data in multiple caches. Transactional Memory (TM) has emerged as a powerful notion which allows an effective concurrency management [9, 13]. A transaction is a block of computations that presents itself as being atomic and executed in isolation. TM systems ease multithreaded application development by allowing the programmer to define that some regions of code, called transactions, must be executed atomically.

In order to make the system highly efficient, TM systems implement concurrent execution of multiple transactions concurrently. In case of a conflict, some of the transactions may be aborted or stalled for some time. A conflict occurs if two or more transactions access to the same memory location

and at least one of the accesses is a write. Transactional Memory (TM) systems must track the read and write sets to detect conflicts among concurrent transactions. An important aspect of conflict detection is recording the addresses that are read or written during a transaction at some granularity (e.g., memory block/word). Some TMs use signatures to summarize unbounded read/write sets in bounded hardware at a performance cost of occurrence of false positives (detection of a conflict that is not present). Each signature is implemented with a single k-hash function Bloom filter (True Bloom signature). Led by Bulk [6], several systems including LogTM-SE [31] and SigTM [18], have implemented read/write sets with per-thread hardware signatures built with Bloom filters [1]. ROCK was a multithreading, multicore, SPARC microprocessor developed at Sun Microsystems which was the first processor to support Transactional Memory in Hardware [7].

B. Perfect Hashing Hardware

Perfect Hashing is also being used to improve the performance in hardware. Perfect Hashing can be used to develop hardware-friendly applications to improve the overall performance of a system. Networking applications use perfect hashing for faster lookup. Some of the applications of perfect hashing has already been explained in the earlier section.

VI. UNIVERSAL HASH FUNCTIONS

Carter and Wegman introduced Universal Classes of Hashing Functions which consisted of classes of hash functions [4]. H_1, H_2 and H_3 class of universal class functions were introduced. Among these H_1 and H_2 class of functions are applicable to file structures and H_3 class of hash functions can be used for hardware implementation.

A. H_1 Class of Hash Functions

H_1 class of hash functions can be defined as follows:

Let $K = \{x_1, x_2, \ldots, x_n\}$ be the given set of keys and $M = \{0, 1, \ldots, m-1\}$ be the memory range available. H_1 class of hash functions is a composition of 2 hash functions $f_{c,d}(x)$ and $g(x)$ defined as

$f_{c,d}(x) = (cx + d) mod\ p$ where p is a prime number, $c > 0, d \geq 0$ and $c, d < p$

$g(x) = x\ mod\ m$

$H_{c,d}(x) = g(f_{c,d}(x)) = ((cx + d) mod\ p) mod\ m$

B. H_3 Class of Hash Functions

H_3 class of hash functions can be defined as follows:

Let Q denote the set of all $i \times j$ Boolean matrices.

For a given $q \in Q$ and $x \in A$, let $q(k)$ be the k^{th} row of the matrix q and and $x(k)$ the k^{th} bit of x. The hashing function $h_q(x): A \rightarrow B$ is defined as

$h_q(x) = x(1).q(1)\ XOR\ x(2).q(2) XOR \ldots\ldots XOR\ x(i).q(i)$

where . denotes the bit by bit AND operation and XOR the exclusive OR operation. The class H_3 is the set $\{h_q | q \in Q\}$. The following example illustrates the hashing functions and hash address calculations. Example: Let i be 8 and j be 3. Then the address space is $A = \{0, 1, \ldots, 255\}$ and the key space is $B = \{0, 1, \ldots, 7\}$. We randomly choose an 8 × 3 matrix q:

$$q = \begin{pmatrix} 1\,1\,0 \\ 0\,0\,1 \\ 1\,1\,1 \\ 1\,0\,1 \\ 0\,1\,1 \\ 1\,0\,0 \\ 0\,0\,1 \\ 0\,0\,0 \end{pmatrix}$$

Then the hash addresses for keys 53 and 100 are

$$h_q(53) = h_q(00110101) = q(3) \oplus q(4) \oplus q(6) \oplus q(8)$$
$$= 111 \oplus 101 \oplus 100 \oplus 000 = 110 = 6(decimal).$$

$$h_q(100) = h_q(01100100) = q(2) \oplus q(3) \oplus q(6)$$
$$= 001 \oplus 111 \oplus 100 = 0100 = 2(decimal).$$

This class of hashing functions is universal. A class H of hashing functions is said to be universal if no pair of keys collide under more than $|H|/m$ of the functions in the class. Here $|H|$ is the number of hashing functions in H and m is the size of the address space.

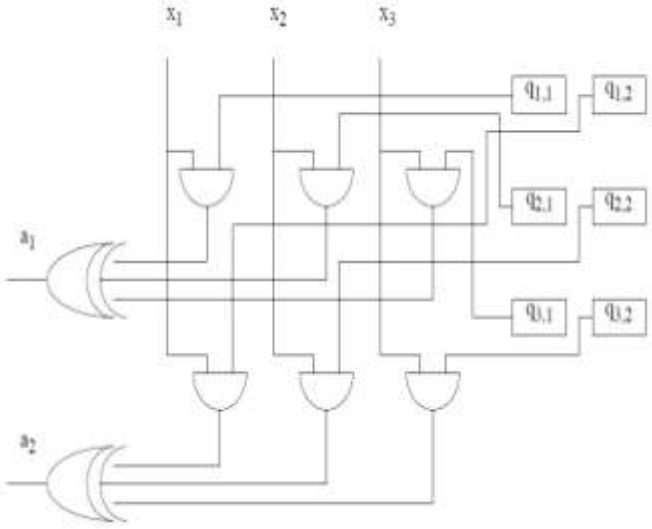

Fig 1. Hardware Hash Address Generator

Hashing functions from this class can be easily implemented in hardware. Figure 1 shows a circuit implementation. When presented with the key $x_1x_2x_3$ the hash address a_1a_2 is the output. The matrix q can be generated in software and then loaded into the bank of registers. The circuit is self-explanatory and we will not elaborate further. Ramakrishna studied the performance of H_3 class of hash functions and concluded that by choosing hashing functions at random from a H_3 class of hashing functions, the performance

in analysis of hashing can be achieved in practice on real-life data [22, 24].

VII. BLOOM FILTER

A Bloom filter is a data structure that is used to check if a given element belongs to a set or not [1]. A search for an element returns a response as 'not in the set' with probability 1 removing the possibility of a false negative. There is possibility of false positive where it responds with 'may be in the set'. Elements can be added to the set, but not removed. For a given vector size, as more elements are added to the set, the probability of false positives increases. Bloom filter was first developed by Bloom in 1970s to separate words based on some predefined rules [1]. He proposed a method to hyphenate words from English dictionary depending on whether they belong to a set or not. Mullin and Margoliash used bloom filter to develop Spell Checking programs which suggested corrections for words that were incorrect [21]. In the earlier days, bloom filter was largely used for database applications to provide faster access to files. Mullin found the use of a bloom filter in predicting the size of a join in a relational database which improved the performance in distributed databases [20]. Originally proposed to reduce the number of accesses to disk, bloom filter finds a number of applications in networking hardware [3, 29]. Some of the applications of Bloom filter in networking includes routing table lookup [32], classification of packets [15], per-flow traffic measurement [14] and so on.

To start with, the Bloom filter is a bit array of m bits, all set to 0. There are k different hash functions, each of which maps or hashes a set element to one of the m array positions with a uniform random distribution. To add an element x, we compute $h_1(x), h_2(x), \ldots h_k(x)$ and set the corresponding bit positions to 1. To query if y is present, we compute $h_1(y), h_2(y), \ldots h_k(y)$ and check to see if the bit is set. If any of the bits at these k positions happens to be 0, then the element is definitely not in the set. If all are 1 then y is in the set with a very high probability.

Table 1. Hash Values for keys x,y,z

Key	h_1(key)	h_2(key)	h_3(key)
x	1	5	13
y	4	11	16
z	3	5	11

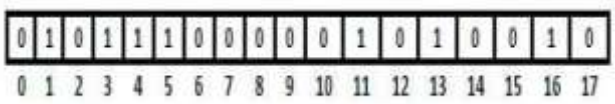

Fig 2. Example for Bloom Vector

Consider the Fig. 2 which shows the structure of a Bloom filter, representing the set x, y, z. The table 1 shows the positions in the bit array that each element is mapped to. For this figure, $m = 18$ and $k = 3$ where m is bloom vector size and k is number of hash functions. For example, consider the element w, which hashes to 4, 13 and 15. As the bit at address 15 is 0, we can conclude that the element w does not belong to the set. For an element v, suppose the hash values are 3, 5 and 11. All the bit positions are 1 and we will falsely conclude that v belongs to the set. It is called a false positive If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive.

VIII. CONCLUSION

Hashing was a very active area of research during 70s and 80s. Since new applications emerged in the areas of Bloom filters and hardware applications, research in this area has invigorated with a large number of publications appearing in recent years. A simple query of "Hashing" to scholar.google.com yielded over 14600 publications in the last two years indicating the current momentum of research in the area. The second author having worked for his Ph.D. thesis over 30 years back is surprised as well as happy for this area to be so active. Given the past, we foresee more applications emerging for hashing techniques and corresponding publications.

REFERENCES

[1] Burton H Bloom. 'Space/time trade-offs in hash coding with allowable errors. Communications of the ACM' , 13(7):422–426, 1970.

[2] Fabiano C Botelho, Nitin Garg, Philip N Shilane, and Grant Wallace. 'Memory efficient sanitization of a deduplicated storage system using a perfect hash function', April 19 2016. US Patent 9,317,218.

[3] Andrei Broder and Michael Mitzenmacher. 'Network applications of bloom filters: A Survey'. Internet mathematics, 1(4):485–509, 2004.

[4] J Lawrence Carter and Mark NWegman. 'Universal classes of hash functions'. In Proceedings of the ninth annual ACM symposium on Theory of computing, pages 106–112. ACM, 1977.

[5] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 'Bulk disambiguation of speculative threads in multiprocessors'. ACM SIGARCH Computer Architecture News, 34(2):227–238, 2006.

[6] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Hoakan Zeffer, and Marc Tremblay. 'Rock: A high-performance sparc cmt processor'. IEEE micro, 29(2):6–16, 2009.

[7] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. 'Extendible hashing, a fast access method for dynamic files'. ACM Transactions on Database Systems (TODS), 4(3):315–344, 1979.

[8] Maurice Herlihy and J Eliot B Moss. 'Transactional memory: Architectural support for lock-free data structures', volume 21. ACM, 1993.

[9] Donald E Knuth. 'The art of computer programming', vol. 3, sorting and searching Addison wesley. Reading, Mass, pages 578–579, 1973.

[10] P.A. Larson. 'Dynamic hashing'. BIT Numerical Mathematics, 18(2):184–201, 1978.

[11] P.A. Larson. 'Linear hashing with separators:a dynamic hashing scheme achieving one access'. ACM Transactions on Database Systems (TODS), 13(3):366–388, 1988.

[12] James R Larus and Ravi Rajwar. 'Transactional memory. Synthesis Lectures on Computer Architecture', 1(1):1–226, 2007.

[13] Tao Li, Shigang Chen, and Yibei Ling. 'Per-flow traffic measurement through randomized counter sharing'. IEEE/ACM Transactions on Networking, 20(5):1622–1634, 2012.

[14] Hyesook Lim and Ha Young Byun. 'Packet classification using a bloom filter in a leaf pushing area-based quad-trie'. In Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on, pages 183–184. IEEE, 2015.

[15] Witold Litwin. 'Linear hashing: a new tool for file and table addressing'. In VLDB, volume 80, pages 1–3, 1980.

[16] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. 'Perfect hashing for network applications'. In 2006 IEEE International Symposium on Information Theory, pages 2774–2778. IEEE, 2006.

[17] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 'An effective hybrid transactional memory system with strong isolation guarantees'. In ACM SIGARCH Computer Architecture News, volume 35, pages 69–80. ACM, 2007.

[18] Michael Mitzenmacher. 'Compressed bloom filters'. IEEE/ACM Transactions on Networking (TON), 10(5):604–612, 2002.

[19] James K. Mullin. 'Optimal semijoins for distributed database systems'. IEEE Transactions on Software Engineering, 16(5):558–560, 1990.

[20] James K Mullin and Daniel J Margoliash. 'A tale of three spelling checkers. Software: Practice and Experience', 20(6):625–630, 1990.

[21] M. V. Ramakrishna. 'Hashing practice: Analysis of hashing and universal hashing'. In Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, SIGMOD '88, pages 191–199, New York, NY, USA, 1988. ACM.

[22] M. V. Ramakrishna and P.A. Larson. 'File organization using composite perfect hashing'. ACM Trans. Database Syst., 14(2):231–263, June 1989.

[23] M.V. Ramakrishna, E. Fu, and E. Bahcekapili. 'Efficient hardware hashing functions for high performance computers'. Computers, IEEE Transactions on, 46(12):1378–1381, Dec 1997.

[24] Celis, Pedro, Per-Ake Larson, and J. Ian Munro. "Robin hood hashing." Foundations of Computer Science, 1985., 26th Annual Symposium on. IEEE, 1985.

[25] M.V. Ramakrishna and GA Portice. Perfect hashing functions for hardware applications. In Data Engineering, 1991. Proceedings. Seventh International Conference on, pages 464–470. IEEE, 1991.

[26] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. ACM SIGCOMM Computer Communication Review, 35(4):181–192, 2005.

[27] Renzo Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. Communications of the ACM, 20(11):841–850, 1977.

[28] G.M. Sridevi, T.V. Rohini, K Kameswari, and M.V. Ramakrishna. 'Bloom vector join for sensor query processing'. In Proc.2nd Internatinal Conference on Computing, Engineering and Information Technology(ICCEIT), Sept, 2013.

[29] Paulus Stravers. 'Translation lookaside buffer', March 18 2004. US Patent 20,040,054,867.

[30] Luke Yen, Jayaram Bobba, Michael R Marty, Kevin E Moore, Haris Volos, Mark D Hill, Michael M Swift, and David A Wood. Logtm-se: 'Decoupling hardware transactional memory from caches'. In High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pages 261–272. IEEE, 2007.

[31] Shijie Zhou and Viktor K Prasanna. 'Scalable gpu-accelerated ipv6 lookup using hierarchical perfect hashing'. In 2015 IEEE Global Communications Conference (GLOBECOM), pages 1–6. IEEE, 2015.

[32] Du, Min Wen, et al. 'The study of a new perfect hash scheme'. IEEE Transactions on Software Engineering 9.3 (1983): 305.

[33] Fredman, Michael L., János Komlós, and Endre Szemerédi. "Storing a sparse table with 0 (1) worst case access time." Journal of the ACM (JACM) 31.3 (1984): 538-544.