An Ad Hoc Adaptive Hashing Technique for Non-Uniformly Distributed IP Address Lookup in Computer Networks

Christopher Martinez
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA

and

Wei-Ming Lin
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA

ABSTRACT

Hashing algorithms have been widely adopted for fast address look-up, which involves a search through a database to find a record associated with a given key. Hashing algorithms transforms a key into a hash value hoping that the hashing renders the database a uniform distribution with respect to the hash value. The closer to uniform hash values, the less search time required for a query. When the database is key-wise uniformly distributed, any *regular* hashing algorithm (bit-extraction, bit-group XOR, etc.) leads to a statistically perfect uniform hash distribution. When the database has keys with a non-uniform distribution, performance of *regular* hashing algorithms becomes far from desirable. This paper aims at designing a hashing algorithm to achieve the highest probability in leading to a uniformly distributed hash result from non-uniform distributed database. An analytical pre-process on the original database is performed to extract critical information that significantly benefits the design of a better hashing algorithm. This process includes sorting the bits of the key to prioritize the use of them in the hashing sequence. Such an ad hoc hash design is critical to adapting to all real-time situations when there exists a changing database with an *irregular* non-uniform distribution.

Keywords: Hashing, Computer Networks, Address Lookup, Packet Matching, Internet, Intrusion Detection, Network Security

1. INTRODUCTION

The computer and communication networks community has seen extensive advancement in its research and commercial fields in the past few decades. To further increase the speed of networks takes more than just physical advances in transmission speed through a given medium. One hindrance that all network components, such as routers, firewalls, intrusion detection, and others, have faced and suffered from it more as the network size grows is from the required search and lookup process through a large address space. Fast address lookup or identification matching has become critical to the feasibility of many modern applications. In a general form, this problem involves a search process through a large database to find a record (or records) associated with a given key. One modern example is in that the routers in wide-area networks have to look through a large database, a routing table, for a forwarding link that matches the given destination address. Another example that calls for imminent attention these days is in the area of internet security, in which intrusion detection demands rapid evaluation of client requests. In this, rules are established to allow the intrusion detection system to check for wrong-doing. Usually, packet headers need to be matched quickly in real time with rule database. Such a matching process usually is carried out through a hashing process to reduce the otherwise potentially excessively long search time.

There have been many schemes developed for IP address lookup problem using a hash function. A complete survey and complexity analysis on IP address lookup algorithms has been provided in [12]. A performance comparison of traditional XOR folding, bit extraction, CRC-based hash functions is given in [4]. Although most of the *regular* hash functions, such as the simple XOR folding and bit extraction, are relatively inexpensive to implement in software and hardware, their performance tends to be far from desirable. CRC-based hash functions are proved to be excellent means but are more complex to compute and implement. Some schemes are hardware based that achieve an improvement in IP look up by maintaining a subset of routing table in a faster cache memory [7,9], while others are software based which improve their search per-

formance mainly through efficient data structures [10,13]. Waldvogel et al. [14] proposed an address look up scheme based on a binary search of hash table, requiring an extra update process in a look up table. Other hashing algorithms have also been widely adopted to provide for the address look-up process [2,3,11,16]. All hashing algorithms inevitably suffer from unpredictable complexities involving conflicts among the data with the same hash result. A search for matching a given query could end up with a sequential search through the number of maximal conflicts in the database. This may result in a long search process time that exceeds the time limitation imposed by design specifications.

When records in the database are key-wise uniformly distributed, any *regular* hashing algorithm would easily lead to the same probabilistically expected optimal performance in terms of search time required. On the other hand, if these records are instead not uniformly distributed, then even different *regular* hash functions would lead to different expected performance. This paper is aimed at the establishment of a methodology following which a hashing algorithm can be designed to achieve the highest probability in leading to a uniformly distributed hash result from a non-uniform database. An analytical pre-process on the original database is first performed to extract critical information that would significantly benefit the design of a better hashing algorithm. This process includes sorting on the bits of the key to prioritize the use of them in the XOR hashing sequence, or a simple bit extraction, or even a combination of both. Significant improvement from our simulation results has been obtained on randomly generated data set as well as real data set.

Hashing for Address Look-up

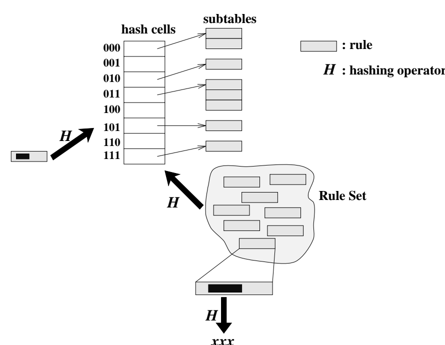


Figure 1: A Hashing Example

Basically, hashing is a process that allows the search to go through a statistically smaller number of steps than a simple sequential straightforward search would have performed. A hash function, usually a mathematical one, maps a number with a large value range into another number with a smaller range. For example, a simplified one as shown in Figure 1, a database of eight given records are to be matched against for any incoming record. Due to the large size of each record and potentially large number of records in the database under real situations, searching through the whole database one at a time could be merely impractical. One may choose to use a portion of the record (or its entirety), as a key, to hash into the target value (a three-bit value as shown in this example) using the hash function (operator) H. Therefore, a database of eight records are now grouped into bins of records according to their corresponding hash results. With this, any incoming record would go through the same hashing to identify the one bin it would need to search through, instead of the whole database. Perfect hashing would guarantee that every bin contains exactly one record, which leads to a search process of exactly one matching to take place. A hash function is considered better than the other if it leads to a smaller expected number of matching steps required. Note that, if the records in the database follow a uniform distribution with respect to value of the key, any *regular* hash function, e.g. non-overlapped XOR, bit extraction, etc., would simply lead to a uniform distribution of these records onto the bins. That is, it should always lead to best expected performance probabilistically in search time required. The goal of this paper is to develop a universal hashing methodology applicable to all non-uniformly distributed data sets.

2. AN HOC ADAPTIVE HASHING FOR DATA SETS WITH AN ARBITRARY *pdf*

One problem that may arise in the lifetime of the database is that, as the number of entries grows and/or the entries are constantly updated, the original hashing algorithm previously designed may become obsolete and leads to a large amount of collisions. To deal with such a situation when there exists a changing database with an *irregular* non-uniform *pdf*, it would be preferable to develop a hashing algorithm based on an ad hoc design that can adapt to the expanding or changing database.

To base a hashing algorithm on an ad hoc design, a parameter must be used to instruct the algorithm to adapt over time. The database is defined as consisting of $M = 2^m$ entries with each entry having n bits in length. In order to render the best (uniform) distribution in the final hashed data set, all the bits in the final hashing function H should demonstrate a distribution as probabilistically random as possible, i.e. evenly distributed between 0's and 1's. An optimal H will have each of its bits demonstrate even distribution of 0's and 1's, and thus leading to the highest probability in reaching the best hashing. The ad hoc parameter used for adaptation in this paper is identified to be the binary distribution of each bit in the n-bit entry key length. For bit position i, d_i is defined as the absolute difference between the number of 0's and 1's in that bit vector across the data set. Once the d value is found for each bit vector as shown in Figure 2(a), they are then sorted into a

non-decreasing order as shown in Figure 2(b). Note that,

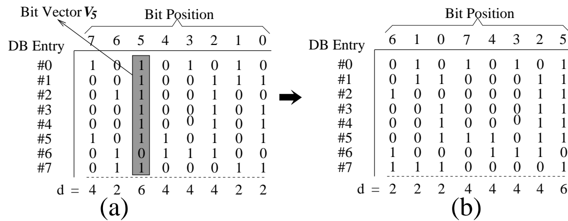


Figure 2: (a) Database with d Values Found (b) Database Sorted By d

while sorting is performed on the d values, no actual rearrangement of the database is needed; instead, an array of the sorted bit indices is used as the hash function. Assume the bit sequence before sorting is $(b_{n-1}, b_{n-2}, \ldots, b_1)$ and the bit sequence afterwards is $(s_{n-1}, s_{n-2}, \ldots, s_0)$. A bit vector V_i with $d_i = 0$ indicate that there are even number of 0's and 1's; while, a $d = M$ reflects that all the bits in the vector have the same value. Translated to the distribution from hashing, a bit of $d = 0$ gives an even hashing distribution (i.e. evenly divided) among the entire hash space while a $d = M$ hashing will result in hashing to only one half of the hash space. Intuitively, using the bits with smaller d values for hashing would lead to a probabilistically better hash distribution, i.e. less conflict in the final mapping. Ideally, if one can identify (or through XOR-ing to obtain) m bits with all their d values equal to 0, it should lead to the best potential performance, assuming no correlation among the bit vectors.

Simulation Setup

Our simulation is devoted to finding the proposed technique on several practical performance indicators. A system is assumed to have a database of 2^m entries each of which contains a key of n bits to be used for hashing into m bits, and thus each such entry is then mapped to a location in a memory of 2^m locations (hashed bins) for matching. Three different performance measurements (indicators) are used to compare different hashing functions: (1) Number of Empty Bins (NEB), (2) the Average maximum Search Length (ASL), and (3) the Maximum Search Length (MSL). An illustrating example of these indicators is given in Figure 3 showing the number of rules mapped (hashed) onto each of the bins. NEB tells how many locations in memory that no rule is mapped into. ASL is the average maximum number of matching steps needed for any given record to match. MSL denotes the largest number of rules that are mapped into any location in the memory. For all of the performance measurements, the smaller the value the better the performance is. The optimal performance value will be 1 for both ASL and MSL, and 0 for NEB, which all indicate that the keys have been evenly distributed among the memory locations. Data sets

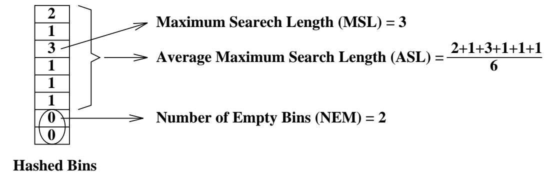
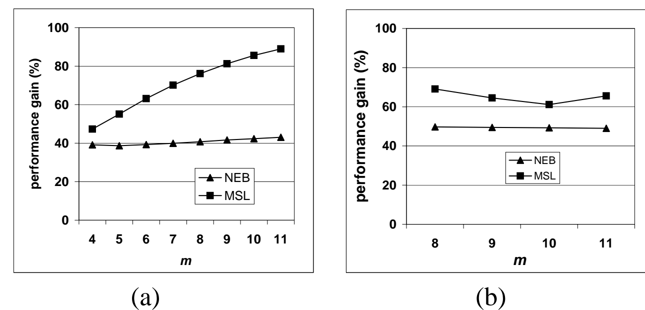


Figure 3: Three Performance Indicators

are generated either randomly or from the real IP addresses collected.

Sorting-Based Bit Extraction Hashing

Bit extraction is a commonly adopted hashing process for its simplicity. Immediate benefit from the sorting process is, when employing simple bit extraction for hashing, significant performance gain can be easily achieved compared to the regular (random) bit extraction process. After the sorting, the first m bits which have the m smallest d values become obvious candidates for bit extraction. That is, the final hash keys are $(s_{n-1}, s_{n-2}, \ldots, s_{n-m})$. From probabilistic point of view, bit extraction using these m bits would lead to the best expected performance among all bit extraction hashing choices. Figure 4 gives the performance comparison between the proposed bit extraction hashing and the random bit extraction approach (i.e. using the first m bits before sorting), when $n = 32$ and m is set to various values. Two data sets are used for simulation,



(a) (b)

Figure 4: Performance Improvement from Bit-Extraction Hashing on (a) Random Data Set (b) Real IP Data Set

one being a data set randomly generated (in (a)) such that the d value for each bit position is uniformly distributed, and the other being the real network traffic subnet ID data set (in (b)). This shows that a very significant improvement margin is obtained by using this simple bit-extraction approach. A close-to-50% reduction in MSL is achieved, thus drastically reducing the search time required by almost one half.

Hybrid XOR Hashing

XOR operator has been widely used for hashing and known to be an excellent operator in enhancing random-

ness in distribution. It also possesses a nice characteristic allowing for analytical performance analysis and thus better algorithm design. XOR-folding is a commonly used hashing technique by simply folding the n-bit key into m-bit hash result through a simple process XORing every $\frac{n}{m}$ key bits into a final hash bit. In some cases, simple extraction may outperform XOR hashing, while in other cases outcome goes the opposite. A combination of both would very likely outperform both if the information from the sorted sequence is properly analyzed and utilized. All-key hashing techniques, i.e. all the bits in the key are used in XORing as in all known XORing hashing techniques, have the benefit in its simplicity, thus easy to implement; however, they tend to over-generalize the data set without fully exploiting the d values from the sorted result. To precisely quantify the benefit in using two bits for XORing with their d values being d_i and d_j, a formula can be derived to find the expected resultant d for the new bit after XORing, denoted as $d_{[d_i \oplus d_j]}$:

$$d_{[d_i \oplus d_j]} = \sum_{k=0}^{x_j} |M - 2\Delta - 4k| \cdot \frac{C_{x_j - k}^{x_i} \cdot C_k^{M - x_i}}{C_{x_j}^{M}} \quad (1)$$
$$\text{where } x_i = \frac{M - d_i}{2},\ x_j = \frac{M - d_j}{2},\ \Delta = x_i - x_j$$

Note that the term $C_{x_j - k}^{x_i} \cdot C_k^{M - x_i} / C_{x_j}^{M}$ indicates the probability for the resultant vector to have a d value equal to $|M - 2\Delta - 4k|$. In order not to degrade the hash performance, every intended XOR operation to be taken between two bits s_i and s_j after sorting, with $d_i \leq d_j$, should lead to a $d_{[d_i \oplus d_j]}$ value such that

$$d_{[d_i \oplus d_j]} < d_i$$

assuming no correlation exists in between bit vectors V_i and V_j; otherwise, simply retaining (extracting) the bit s_i for hashing would yield better performance than using XORing of the two bits s_i and s_j.

A complete spectrum of $d_{[d_i \oplus d_j]}$ for all possible integral values of d_i and d_j for $M = 32$ is given in Figure 5. Figure 6 displays, for each d_i, the cutoff d_j ($\geq d_i$) value that would lead to $d_{[d_i \oplus d_j]} < d_i$, i.e., the smallest d_j that can still be a candidate to be XORed with d_i for better performance. For example, when $M = 64$, a $d = 0$ will never achieve any more improvement when XORing with any d value, i.e. its cutoff is $d_j = 64$. On the other hand, a $d = 6$, with its cutoff equal to $d_j = 36$, is allowed to XORed with a bit with its d value at least equal 36. Note that, for hashing a database, multiple bits (i.e. more than two bits) can be XORed together. That is, when more than two bits are to be XORed, their final d value would require some additional process. It can be shown that the process in determining the d value among three bits is a process with the property of associativity, that is,

$$d_{[d_{[d_i \oplus d_j]} \oplus d_k]} = d_{[d_i \oplus d_{[d_j \oplus d_k]}]} \quad (2)$$

Note that, $d_{[d_i \oplus d_j]}$ (or $d_{[d_j \oplus d_k]}$) may not be an integer anymore, thus the formula in Equation 1 is no longer applicable. We have chosen a simple "interpolation/extrapolation"

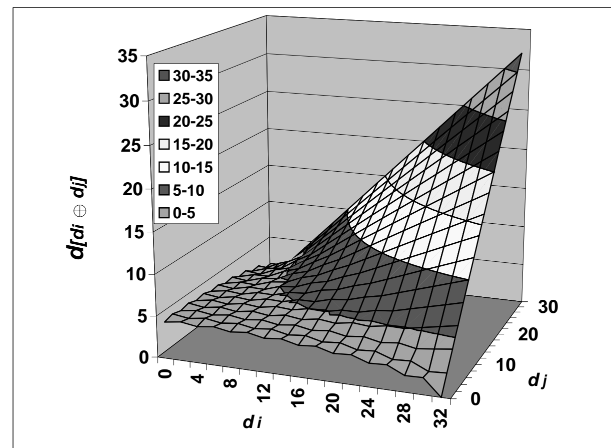


Figure 5: Spectrum of $d_{[d_i \oplus d_j]}$ Values

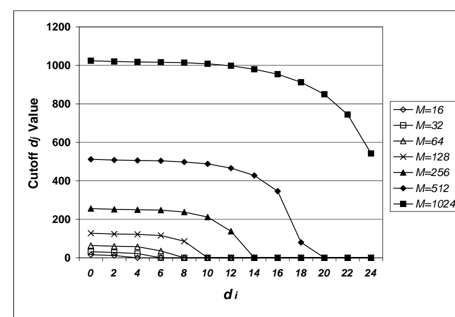


Figure 6: Cutoff Points for Each Integral d_i for Different M

approach to approximate the result when non-integral d values are involved when using the formula in Equation 2.

After the bit vectors are sorted with their d values, our proposed *hybrid hashing* technique is to adopt a *folding* hash approach in finding suitable candidate bits to XOR. During each folding cycle, *at most* m more bits from the right end of the sorted bit sequence are XORed with the current m-bit intermediate hash result with their current d values denoted as $(D_{m-1}, D_{m-2}, \ldots D_0)$. Initial hash result is simply an extraction of the first m bits of the sorted sequence, thus,

$$(D_{m-1}, D_{m-2}, \ldots D_0) = (d_{n-1}, d_{n-2}, \ldots, d_{n-m})$$

Groups of bits hashed so far are recorded as $(G_{m-1}, G_{m-2}, \ldots, G_0)$, and it is initialized to the first m bits:

$$(G_{m-1}, G_{m-2}, \ldots, G_0) = (\{s_{n-1}\}, \{s_{n-2}\}, \ldots, \{s_{n-m}\})$$

Essentially, in each folding cycle, each intermediate hashing bit-group result with an intermediate d value equal to D_i, starting with the one with the smallest d value, is XORed with the rightmost bit s_j with a d value of d_{s_j} if $d_{[D_i \oplus d_{s_j}]} < D_i$; otherwise it has reached its cutoff point. Folding continues until no more bits are left for XORing. Once cutoff point is reached for an intermediate hash bit, it is automatically excluded for further XORing in subsequence folding cycles since the sequence is already a sorted one. The complete algorithm is shown in Figure 7. For example, given a database with 16 entries with each entry 12 bits long, the database key will be hashed into four bits. An example is given in Figure 8. In this example, the d values for the bit vectors are $\{2, 2, 4, 4, 6, 6, 8, 8, 8, 10, 10, 14\}$ after sorting. Initial hash result is simply $(D_3, D_2, D_1, D_0) = (2, 2, 4, 4)$ and the bit-grouping is $(G_3, G_2, G_1, G_0) = (\{s_{11}\}, \{s_{10}\}, \{s_9\}, \{s_8\})$. In the first folding cycle, bit s_0 is XORed with bit s_{11} since $d_{2\oplus 14} = 1.75 < 2$, while for bit s_{10}, the next candidate, bit s_1 has a d value of 10 already passing the cutoff point of $d_{s_{10}} = 2$. Bits s_9 and s_8 subsequently take their corresponding XOR candidates, bit s_1 and s_2, to wrap up the first folding cycle with $(D_3, D_2, D_1, D_0) = (1.75, 2, 3.36, 3.36)$ and $(G_3, G_2, G_1, G_0) = (\{s_{11}, s_0\}, \{s_{10}\}, \{s_9, s_1\}, \{s_8, s_2\})$. In the second folding cycle, D_3 runs into its cutoff from $d_{s_3} = 8$, and therefore bit s_3 is XORed with D_1 and bit s_4 is XORed with D_0, ended with $(D_3, D_2, D_1, D_0) = (1.75, 2, 2.99, 2.99)$ and $(G_3, G_2, G_1, G_0) = (\{s_{11}, s_0\}, \{s_{10}\}, \{s_9, s_1, s_3\}, \{s_8, s_2, s_4\})$. In the third folding cycle, all D's have their cutoff met and thus ending the process. Note that, this greedy algorithm requires a very economic pre-processing time complexity of $O(n)$ in terms of the number of *bits* to visit, while an optimal algorithm using an exhaustive search would require a time complexity of $O((2^n)^m)$ by allowing all the n bits to potentially contribute to each of the m hash bits.

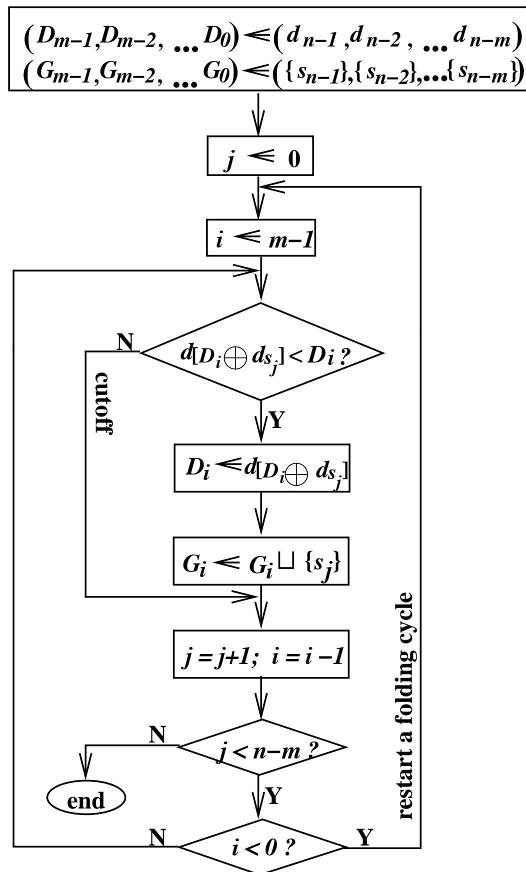


Figure 7: The Algorithm to Determine a Hybrid XOR Hashing Function

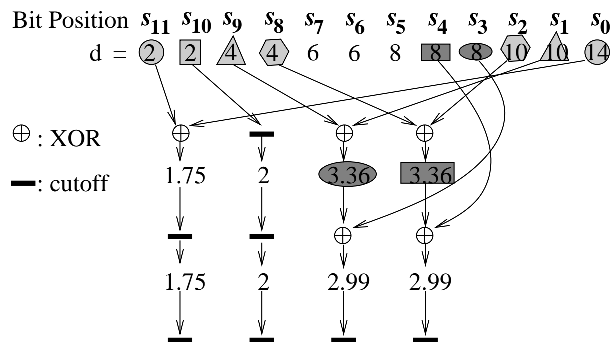


Figure 8: An Example of the Hybrid XOR Process

Figure 9 shows the performance improvement from using the proposed Hybrid XOR approach over the straightforward all-key XOR on random data set, and Figure 10(a) shows the performance improvement on a real IP data set. Figure 10(b) gives breakdowns of MSL improvement on

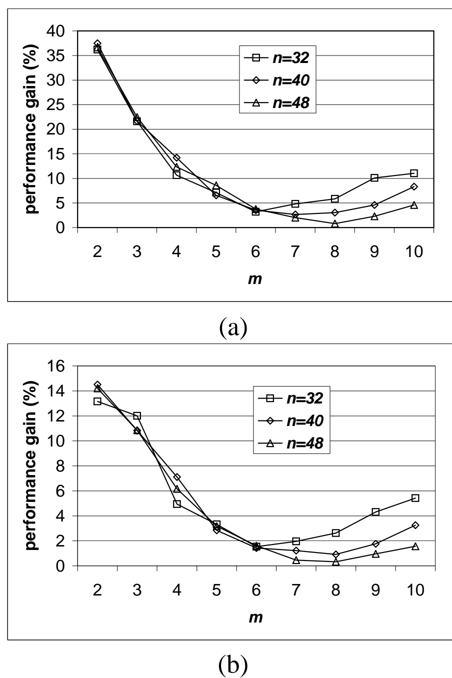


Figure 9: Performance Improvement in (a) MSL and (b) ASL with Hybrid XOR on Random Data Set

three different types of IP addresses. Very respectable gains are reached in both results. Performance improvement is very significant when m is small, that is, when more folding is allowed. When m becomes larger, the proposed ad hoc algorithm gradually loses its advantage until m passes a threshold value. This is most likely due to the fact that, on MSL, the original XOR-folding approach is incapable of limit the search length (the maximal number of data mapped to the same bin) as well as the proposed technique when m is increased. From (b), type-A IP addresses lead to more improvement with the proposed technique than the other two types. This could be due to the scenario that more subnet addresses in this type are more aggregated (i.e. imbalanced) than the other two types,

3. CONCLUSION

The wide applicability of the proposed methodology is clearly demonstrated by our preliminary application results. General applications include: (1) IP Address Lookup (2) Intrusion Detection Systems (3) General Database Query (4) String Matching. The success of this development will benefit many more application engineers, scientists and system designers/analysts. There are still many potential extensions along this line of research, including:

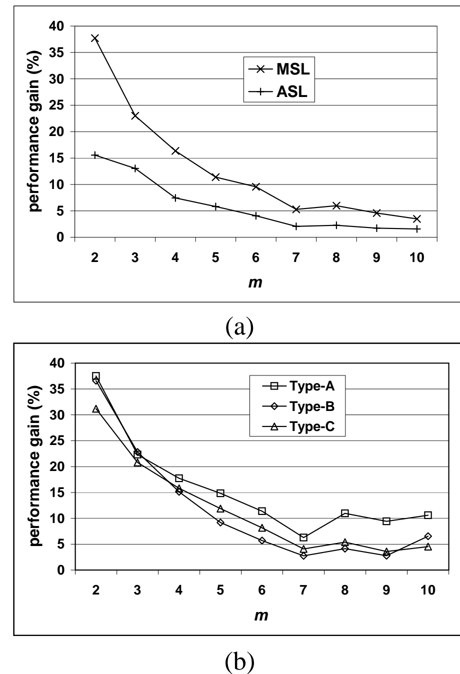


Figure 10: Simulation Results of Improvement Using Real Work Loads with $n = 32$: (a) Aggregate Results (b) Breakdown on Types

- Precise calculation of d values: we plan to extend our analytical work to find an even more reliable approximation of the d value calculation when the composing d values are not integers.
- Effect of d values and their combinations on performance: Exact effect on performance from a hashing with a group of final d values still remains unsolved. For example, for $m = 2$ (or even larger m), between two final hashing results, (D_1, D_0) and (D'_1, D'_0), if $D_1 < D'_1$ but $D_0 > D'_0$, how to determine on the better one requires more a complex analysis and the analysis becomes harder when m becomes larger. A thorough understanding of this is critical to the finding of the best hashing function.
- Correlation among bit vectors: when there exists correlation among bit vectors as most real data sets exhibit, the best hashing produced by relying solely on the d values may turn out to be far from the best. How to come up with a standard and reliable calibration mechanism to quantify the correlation between the two bit vectors (c_{ij}) and derive a function $F(c_{ij}, d_i, d_j)$ to decide whether or not the intended XOR is beneficial.
- Effect of bit re-using: Once bit(s) are re-used, correlation between bit positions sharing the bit(s) arises. How to decide on whether the ensuing correlation is

too big an investment to compensate from the gain through using the misleadingly smaller d values remains a challenge for this project.

4. REFERENCES

[1] Ole Amble and Donald E. Knuth, "Ordered Hash Tables. Chapter 7 in D.E.Knuth Selected Papers on Analysis of Algorithms," *CSLI Stanford CA,*, 2000.

[2] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups", *IEEE INFOCOM*, 2001.

[3] Sang-Hun Chung, J. Sungkee, Hyunsoo Yoon, Jung-Wan Cho, "A Fast and Updatable IP Address Lookup Scheme", *International Conference on Computer Networks and Mobile Computing*, 2001.

[4] Raj Jain, "A Comparison of Hashing Schemes for Address Lookup in Computer Networks," *IEEE Transactions on Communications,*, Vol. 40, No. 10, Oct 1992.

[5] Donald E. Knuth, "The Art of Computer Programming," *Vol 3: Sorting and Searching. 2nd Ed., Addison-Wesley, Reading MA*, 1998.

[6] Christopher Martinez, Wei-Ming Lin and Parimal Patel, "Optimal XOR Hashing for A Linearly Distributed Address Lookup in Computer Networks", *Symposium on Architectures for Networking and Communications Systems*, Oct., 2005, Princeton, New Jersey

[7] Andreas Moestedt, Peter Sjodin, "IP Address Lookup in Hardware for High-speed Routing", *Proc. IEEE Hot Interconnects 6 symposium*, Stanford, California, pp.31-39, August 1998.

[8] P. Newman, G. Minshall, T. Lyon, and L. Hutson, "IP Switching and Gigabit Routers," *IEEE Communications Magazine*, January 1997, p.64-p.69.

[9] Xiaojun Nie, David J. Wilson, Jerome Cornet, Gerard Damm, Yiqiang Zhoa, "P Address Lookup Using A dynamic Hash Functio", *IEEE Electrical and Computer Engineering, Canadian Conference*, Page(s) 1646 - 1651, May 1-4, 2005.

[10] Stefan Nilsson and Gunnar Karlsson, "IP Address Lookup Using LC-Tries", *IEEE Journal on Selected Areas in Communications*, pp. 1083-1092, June 1999.

[11] D. Pao, C. Liu, L. Yeung and K.S. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup", *IEEE INFOCOM*, 2002.

[12] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms", *IEEE Network*, Vol.15, pp.8-23, Mar./Apr.2001.

[13] V. Srinivasan and G. Varghese, "Faster Ip Lookups Using Controlled Prefix Expansion", *Proceedings of SIGMETRICS 98*, pp. 1-10, Madison, 1998.

[14] M. Waldvogel, G. Varghese, J. Turner and B. Plattner, "Scalable High Speed Ip Routing Lookups", in *Proc. ACM SIGCOMM'97*, pp. 25-35, Sept. 1999.

[15] G.B. White and M.I. Huson, "A Peer-based Hardware Protocol for Intrusion Detection Systems", *Military Communications Conference (MILCOM)* 1996.

[16] P.A. Yilmaz, A. Belenkiy, N. Uzun, N. Gogate and M. Toy, "A Trie-based Algorithm for IP Lookup Problem", *Global Telecommunications Conference (GLOBECOM)* 2000.

[17] Daxiao Yu, Brandon Smith, and Belle Wei, "Forwarding Engine for Fast Routing Lookups and Updates," *Global Telecommunications Conference,*, Globecom '99, p.1556-p.1564.