DEC-TR-593

A Comparison of Hashing Schemes for Address Lookup in Computer Networks

Raj Jain

Digital Equipment Corporation
550 King St. (LKG1-2/A19)
Littleton, MA 01460

Raj Jain is now at
Washington University in Saint Louis
Jain@cse.wustl.edu
http://www.cse.wustl.edu/~jain/

Network Address: Jain%Erlang.DEC@DECWRL.DEC.COM

February 1989
This report has been released for external distribution.

A Comparison of Hashing Schemes for Address Lookup in Computer Networks

Raj Jain
Distributed Systems Architecture & Performance
Digital Equipment Corp.
550 King St. (LKG 1-2/A19)
Littleton, MA 01460
ARPAnet: Jain%Erlang.DEC@DECWRL.DEC.COM

DEC-TR-593

Version: April 12, 1989

Abstract

The trend toward networks becoming larger and faster, and addresses increasing in size, has impelled a need to explore alternatives for fast address recognition. Hashing is one such alternative which can help minimize the address search time in adapters, bridges, routers, gateways, and name servers.

Using a trace of address references, we compared the efficiency of several different hashing functions and found that the cyclic redundancy checking (CRC) polynomials provide excellent hashing functions. For software implementation, Fletcher checksum provides a good hashing function. Straightforward folding of address octets using the exclusive-or operation is also a good hashing function. For some applications, bit extraction from the address can be used. Guidelines are provided for determining the size of hash mask required to achieve a specified level of performance.

1 INTRODUCTION

The trend toward networks becoming larger and faster, addresses becoming larger, has impelled a need to explore alternatives for fast address recognition. DECnet Phase IV currently allows upto 64,000 nodes and DEC's internal network called EasyNet [21] already has more than 30,000 nodes. Such large networks obviously need more efficient address lookups. The size of the addresses themselves is also growing. HDLC, a commonly used datalink protocol standard, was designed with 8-bit addresses. All IEEE 802 LAN protocols and Ethernets support 48-bit addresses while the ISO/OSI network layer requires 160-bit (20 octets) addresses. This increased length of the search key has also necessitated a need to find efficient ways to look up addresses. Finally, because networks are becoming faster, network routers, which previously handled a few hundred packets per second are now expected to handle 8000 to 16,000 packets per second. This fast handling requires squeezing every cycle out of the frame forwarding code.

The organization of this paper is as follows. In the next section, we describe a number of problems in networking design that require searching through a large database. In Section 3, we discuss a number of possible solutions including caching and hashing. In a companion paper [13], we compared the performance of various cache replacement algorithms. One of the unexpected results of this analysis was that in some cases, caching could be harmful in the sense that the performance would be better without caching. We, therefore, tried hashing as a possible solution to the problem of fast searching through the address database. After a brief introduction to hashing concepts, we develop a metric to compare various hashing functions. We then use the trace data to compare several different hashing functions.

2 A General Problem

One of the performance problems encountered repeatedly in computer systems design is that of searching through a large information base. Simply stated, the problem is that of finding the information associated with a given key. High performance access to information is particularly interesting if the number of keys is large or if the time to access the main information base is long as is the case if the information is located remotely. Some of the areas in the design and implementation of computer networks where this problem is encountered are as follows:

Datalink adapters on local area networks (LAN) need to recognize the destination addresses of frames on the LAN. Most adapters have only one physical address, which can be easily recognized. However, each station also accepts a number of multicast addresses and the adapter must quickly decide whether to receive a multicast frame. In some token ring networks, e.g., Fiber Distributed Data Interface (FDDI) [6,22], stations need to set an *address recognized* flag in the frame. For the smallest size frames this means that the address has to be recognized within 13 octets (1.04 μs). This puts an upper bound on the time within which end stations have to recognize the multicast addresses they want to listen to.

Bridges, used to interconnect two or more LANs, have to recognize the destination addresses of every frame and decide quickly whether to receive the frame for forwarding. In order to learn the relative locations of stations, transparent learning bridges [7] need to recognize source addresses also.

Routers in wide area networks (WAN) have to look through a large forwarding database to decide the output link for a given destination address.

Several high-speed networks simplify the problem of address lookup by using a hierarchical address format that allows the forwarding path to be looked up directly. Although it does make the routing fast, association of a destination's unique identifier (generally a 48-bit physical address) to its hierarchical address at the originating station still requires searching through a large address database.

Name servers have the ultimate responsibility for associating names to characteristics. Among all the applications listed here, name servers probably have the largest information base and the problem is most acute.

In all of the above applications, time to search through a large information base has a significant impact on the overall performance and an analysis similar to that presented here would be helpful in improving the performance.

3 Possible Solutions

The time to access information is a function of several parameters, including the following:

1. Size of the information base
2. Usage pattern
3. Key structure
4. Storage structure
5. Storage location
6. Access method

To make the access more efficient we need to consider changing each one of the above six parameters. The first parameter, the size of the information base, is really not under the control of the system designers. In the future, the size is only going to grow and make the problem worse. We, the system designers, have only indirect control, if any, over the second parameter, the usage pattern. By rewarding certain usage patterns, for example, by providing a faster response to these patterns, we can encourage users to follow certain patterns. The key to efficient access lies in the remaining four parameters.

By properly organizing key structures, e.g., with hierarchical addresses, we can partition the information base into manageable chunks. Most large networks have several levels of hierarchy. DECnet Phase IV, for example, has two levels of hierarchy. The network consists of several areas each with up to 1024 end stations.

The second way to solve the problem is to organize the storage into several levels of hierarchy. For example, most frequently used addresses could be kept in a cache. Addresses not found in the cache would be looked up in the full database. This is a two-level hierarchy. An obvious extension is an n-level hierarchical storage structure in which addresses not found in ith level are then looked up in $(i+1)$th level. Caching is particularly helpful if the reference pattern has a locality property [11].

In some cases, the problem is solved by locating different levels of the storage hierarchy at successively more remote locations. For example, the clients of a name server could keep a local copy of the frequently used names. This is also called caching. In this case, the difference between access time to local copy and remote database is so different that caching can be justified even if there is very little locality in the usage pattern.

Finally, the time to access can be reduced by devising efficient search strategies. Various searching methods, such as tree and trie search strategies, have been developed to efficiently find a key in a table of keys [25]. One method, which we analyze in this paper, is hashing. If properly designed, a hashing algorithm can allow a very large information base to be searched in constant time. In fact, hashing is already being used in an existing LAN adapter to recognize multicast addresses.

4 Measured Environment

In order to compare various hashing strategies, we used a trace of destination addresses observed on an extended local area network in use at Digital's King Street, Littleton facility. The network consists of several Ethernet LANs interconnected via bridges. The network is a part of Digital's company-wide network called EasyNet [21], which has more than 30,000 nodes. The building itself has approximately 1200 stations on several Ethernet LANs interconnected via approximately 80 bridges. A number of routers connect the extended LAN to the rest of the Easynet. There are 30 Level 1 routers and 6 Level 2 routers in the building. A promiscuous monitor attached to one of the Ethernet LANs produced a time-stamped reference string of 2.046 million frames observed over a period of 1.09 hours. A total of 495 distinct station addresses were observed in the trace, of which 296 were seen in the destination field. Due to bridge filtering, only those frames whose desinations have a short path through the monitored segment are seen on the segment.

There are several advantages and disadvantages to using a trace. A trace is more credible than references generated randomly using a distribution. On the other hand, traces taken on one system may not be representative of the workload on another system. We hope that others will find the methodology presented here useful and will apply it to traces taken in environments relevant to their applications.

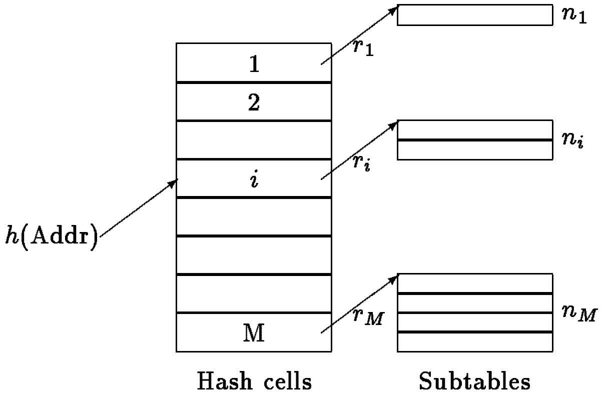


Figure 1: Hashing concepts.

5 Hashing: Concepts

Webster's dictionary defines the word 'hash' as a verb *"to chop (as meat and potatoes) into small pieces"* [31]. Strange as it may sound, this is correct. Basically, hashing allows us to chop up a big table into several small subtables so that we can quickly find the information once we have determined the subtable to search for. This determination is made using a mathematical function, which maps the given key to hash cell i, as shown in Figure 1. The cell i could then point us to the subtable. We will use n_i to denote the size of the ith subtable and M to denote the number of hash cells. Ideally, one would like to use a hashing function so that each subtable has only one entry so that no further searching or subtables are required. For most hashing functions, the size of subtables n_i decreases as the size of the hash table M increases. For an very large number of hash cells, one is almost guaranteed to be able to find the desired key without further search.

For finite hash tables, two or more keys may map to the same hash table location leading to a *collision.* Most of the hashing literature is about what to do after a collision. If the hash table size is larger then or equal to the total number of keys, one does not need subtables and use the hash table itself to store the keys and other information. Several techniques, such as *linear probing* and *double hashing*, have been devised to resolve the collisions in as few attempts as possible. *Dynamic hashing* schemes allow the table size to increase dynamically as the number of entries grows [4,16]. *Perfect hashing* schemes also exist, which cause no collisions [3,30]. *Minimal perfect hashing* functions not only avoid collisions, but also leave no empty space in the hash table [1,2,10,23]. For surveys of various hashing schemes and issues see [14,15,17,18,19,20,25,27].

If the hash table size is less than the total number of keys, collisions are unavoidable. We would like the hashing function to be such that the addresses which are looked up more often are in smaller subtables. It is desirable to minimize the average number of lookups required for the trace. To compute this, we define the following symbols:

R = Number of frames in the trace
N = Number of distinct addresses in the trace
M = Number of hash cells
= Number of subtables
n_i = Number of addresses that hash to ith cell
$\sum_i n_i = N$
r_i = Number of frames that hash to ith cell
$\sum_i r_i = R$
p_i = Fraction of addresses that hash to ith cell
$= \frac{n_i}{N}$
q_i = Fraction of frames that hash to ith cell
$= \frac{r_i}{R}$

If we perform a regular binary search through all N addresses, we need to perform $1+\log_2(N)$ or $\log_2(2N)$ lookups per frame. Given an address that hashes to ith cell, we have to search through a subtable of n_i entries. Using a binary search, we would need only $\log_2(2n_i)$ lookups. The tot!l number of lookups is:

$$\text{Number of lookups for the trace } = \sum_i r_i(\log_2(2n_i))$$

$$\text{Number of lookups per frame } = \frac{1}{R}\sum r_i(\log_2(2n_i))$$

Compared to $\log_2(2N)$ lookups per frame, the net saving due to hashing is:

$$\begin{aligned} &\text{Lookups saved per frame} \\ &= (\log_2(2N)) - \sum_i \frac{r_i}{R}(\log_2(2n_i)) \\ &= \sum_i -\frac{r_i}{R}\log_2(\frac{n_i}{N}) \\ &= \sum_i -q_i \log_2(p_i) \qquad (1) \end{aligned}$$

Here, q_i and p_i are probabilities such that $\sum_i q_i = 1$ and $\sum_i p_i = 1$. The goal of a hashing function is to maximize the quantity $\sum -q_i \log_2(p_i)$. Notice that p_i and q_i are not related. In the special case of all addresses being equally likely to be referenced, q_i is equal to p_i and the expression $\sum -p_i \log_2(p_i)$ would be called the **entropy** of the hashing function. It is because of this similarity that we will call the quantity $\sum -q_i \log_2(p_i)$ the entropy or **information** content of the hashing function. It is measured in units of 'bits.' We illustrate its computation using a simple example.

Table 1: Computing Information in the Last Two Bits

Bits	# of Frames	# of Addr.	q_i	p_i	$-q_i \log_2 p_i$
00	1252479	239	0.61	0.48	0.65
01	219989	71	0.11	0.14	0.31
10	148725	55	0.07	0.11	0.22
11	424807	130	0.21	0.26	0.41
$\sum$	2046000	496	1.00	1.00	1.59

Hashing is usually performed in two steps. In the first step, an address A is converted to a hash value $f(A)$. In the second step, some m bits of $f(A)$ are extracted so that the total number of hash cells is 2^m. For example, one could take the last m bits of $f(A)$:

$$h(A) = \text{Mod}\{f(A), 2^m\}$$

Here, $f(A)$ is usually a complex operation. In the simplest case, we could have $f(A) = A$ and take the last two bits, for instance, of the address as our hashing function. This will break the address table into four subtables. The number of address entries in these four subtables and the corresponding number of frames refering to these subtables using our measured trace are shown in Table 1. The ratio of the number of frames that refer a subtable and the total number of frames gives the probability q_i. Similarly, the ratio of the number of addresses in a subtable to the total number of addresses gives the probability p_i. The information entropy for each subtable is then computed and added to give the total entropy for this hashing function. For our trace, we found that the last two bits of the address have an entropy of 1.59 bits. In other words, if we use the last two bits of the address to decide which subtable to search, we would save 1.59 lookups per frame.

We do not have to limit ourselves to the last two bits. We could use any two consecutive bits i and $i+1$. The resulting information as a function of i is shown in Figure 2. Here, the most significant bit of the address is denoted as the 0th bit, and the least

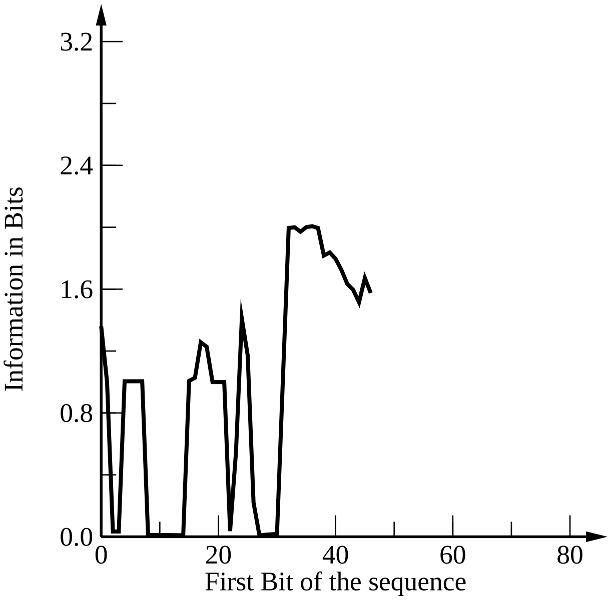


Figure 2: Information in two consecutive bits of addresses.

significant bit is the 47th bit. Notice that the best two bits for this trace are bits 36 and 37. These provide a full two bits of information, the maximum that one could expect of two bits.

6 Hashing Using Address Bits

We can now generalize the concept of hashing using address bits to m bits with $m = 1, 2, \ldots$. In general, we could hash using bits i, $i+1$, ..., $i+m-1$ of the address to 2^m subtables. The number of lookups saved, as we saw in the last section, is equal to the information entropy of the bits. For our trace, this is plotted in Figure 3. The starting position i of the bit sequence is plotted along the horizontal axis and the information in bits i through $i+m-1$ computed using Equation 1 is plotted along the vertical axis. Eight curves corresponding to m consecutive bits with $m = 1, 2, \ldots, 8$ are plotted.

From Figure 3, we observe that:

1. All 8-bit sequences between bits 0 and 24 have less than two bits of information.
2. The bits 32 through 39, in general, have a high information content.

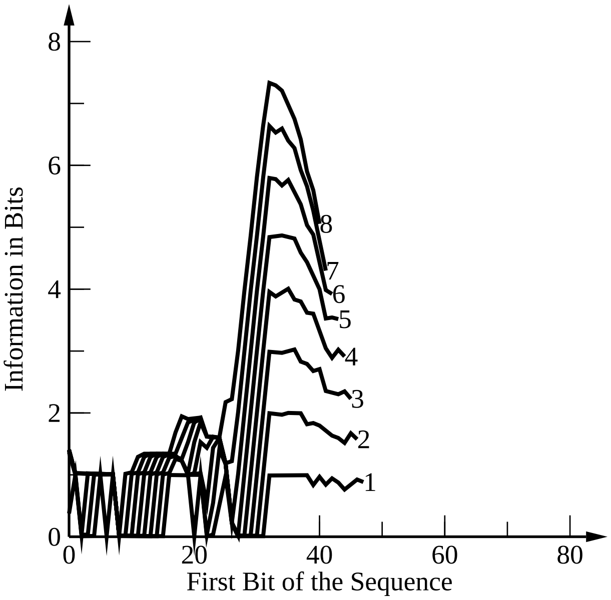


Figure 3: Information in address bits.

The first observation is not surprizing considering that the first three octets of the IEEE 802 addresses used on IEEE 802 LANs are assigned by the IEEE and, thus, most stations have the same first three octets. The first two bits corresponding to the global/local assignment and multicast/unicast addresses may be different in different addresses. Given these two bits, other bits can be easily predicted. In multivendor environments the first 3 octets may have a little more information. However, in general, these bits are not good for hashing purposes.

The second observation says that the fifth octet of the address has the highest information *in our environment.* This observation, if applicable, leads to the following types of conclusions:

1. *Use the fifth octet as the hashing function.* If one were hashing solely based upon the address bits, the bits in this octet would provide a maximum savings in the number of lookups. If one were using this hashing function to filter out unwanted frames, these bits would provide the least number of them.
2. *When comparing two addresses, compare the fifth octet first.* If the addresses are not equal, the very first comparison will fail more often than when using other octets.
3. *Use the fifth octet as the branching function at*

the root of a tree database. If the addresses are stored in a tree [28] or trie structure [25] and the address bits are used to decide the branch to be taken, using bits from this octet would provide maximum discrimination. Compare this to using the bits from the first three octets. Most of the bits in the first three octets are the same in all addresses and we would end up following the same first step for all searches.

4. *Use the fifth octet as the load balancing function for different paths.* Given several alternative paths to a set of destinations, one way to balance the load on different paths is to select a path randomly. This causes out-of-order arrivals at the destination. A better alternative is to decide the path based on a few bits of the address. This ensures that all frames going to a particular destination follow a single path and load balancing is achieved by different destinations using different paths.

It should be obvious that the fifth octet may not be the most informative octet for all environments. Nonetheless, the above recommendations are useful providing that one uses the appropriate, most informative octet instead.

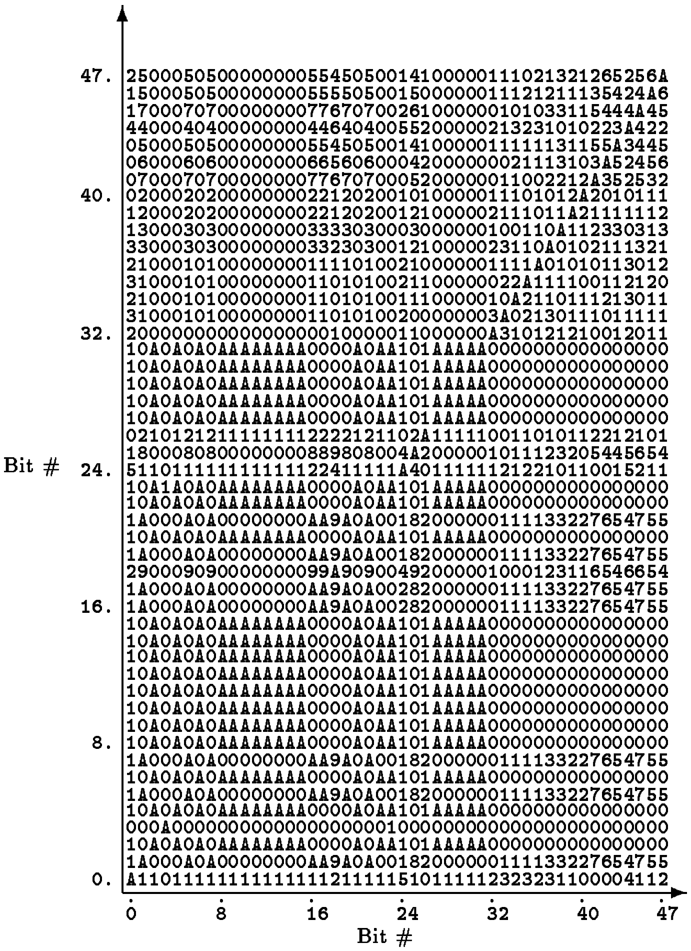
7 Correlation of Address Bits

In the last section we saw that many bits in the address had much less than one bit of information. Even though many bits may have one bit of information each, the combined information of these bits may not be much. This is due to a correlation among bits. Statistically, the correlation between two random variables x and y is computed as follows:

$$\text{Correlation}(x,y) = \frac{E[(x-\bar{x})(y-\bar{y})]}{\sqrt{E[(x-\bar{x})^2]E[(y-\bar{y})^2]}}$$

Here, $\bar{x}$ is the mean of x. The correlation always lies between -1 and $+1$.

If two bits are independent, their correlation is zero and the combined information of the two bits is equal to the sum of their individual information. On the other hand, if two bits are related such that one is always identical to the other or one is always a complement of the other, their correlation is ±1, and the combined information content of the two bits is the same as that of one of the bits. Thus, when using several bits from the address, one should choose bits that have the least correlation.



```
        47. 250005050000000055450500141000001110213212 65256A
```

Figure 4: Correlation among address bits.

Using our trace, we measured correlation between different bits. The results are shown schematically in Figure 4. The figure shows a 48 × 48 matrix of correlation. Only the first digit after the decimal point is shown and the sign is omitted. For example, a correlation of ±0.6 is shown as 6. A correlation of +1 or −1 is shown by the letter 'A.' From this figure, we see that bits 1 through 31 are highly correlated to each other. The occurrence of 'A' is limited to these bits. The diagonal of the correlation matrix is always 'A' since every bit is always fully correlated with itself. This explains why combining the first 32 bits did not improve the information.

8 Hashing Using CRC

So far, we assumed that the subtable to be looked up be decided by extracting a few bits from the address. Given that in many LANs, the destination address is the first part of the frame, and the frame is passed through the cyclic redundancy check (CRC) circuit, the bits of the CRC provide another alternative for 'no cost' hashing. This scheme is, in fact, employed in American Micro Device's LANCE chip, used in many Ethernet adapters. Each station has a number of multicast addresses that it wants to receive. The adapter uses the CRC polynomial as a hash function to quickly reject frames with undesired multicast addresses. The chip uses the most significant 6 bits in the CRC shift register as an index into a 64-bit mask. If the 6 bits constitute an index i and the ith bit in the mask is set, the frame is accepted, otherwise it is rejected.

Given an address $\{a_0, a_1, a_2, \ldots, a_{47}\}$, the 32-bit CRC of the address can be computed by forming the following polynomial:

$$a(x) = \sum_{i=0}^{31} \bar{a}_i x^{47-i} + \sum_{i=32}^{47} a_i x^{47-i} \qquad (2)$$

and computing the remainder when the above polynomial is divided by the following CRC polynomial (used in IEEE802 protocols):

$$\begin{aligned} g(x) &= x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + \\ & \quad x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1 \end{aligned}$$

The coefficient of the remainder polynomial can be used as a hashing function. Notice that in Equation 2, the first 32 bits of the address are complemented. For details of this process see [12,8]. Although, an algebraic description of CRC computation as presented above sounds tedious, its hardware implementation using shift registers is straightforward.

The LANCE chip uses the most significant 6 bits of the CRC as a hashing function. We are interested in finding out how well this hashing function works and if we can extend it to more bits. To do this, we computed the information content of m-bit sequences consisting of bits i through $i+m-1$ of the 32-bit CRC for m = 1, 2, ..., 8, and i = 0, 1, ..., 31. Again, $i = 0$ represents the most significant bit of the CRC. The results are shown in Figure 5.

It is interesting to compare Figures 3 and 5. Notice the following:

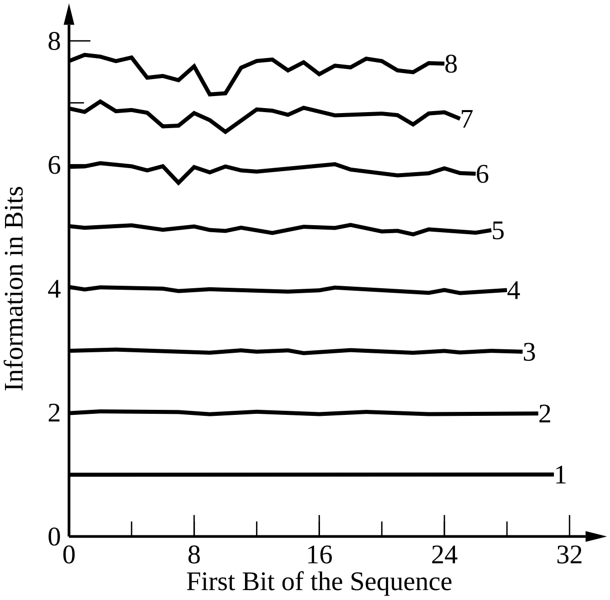
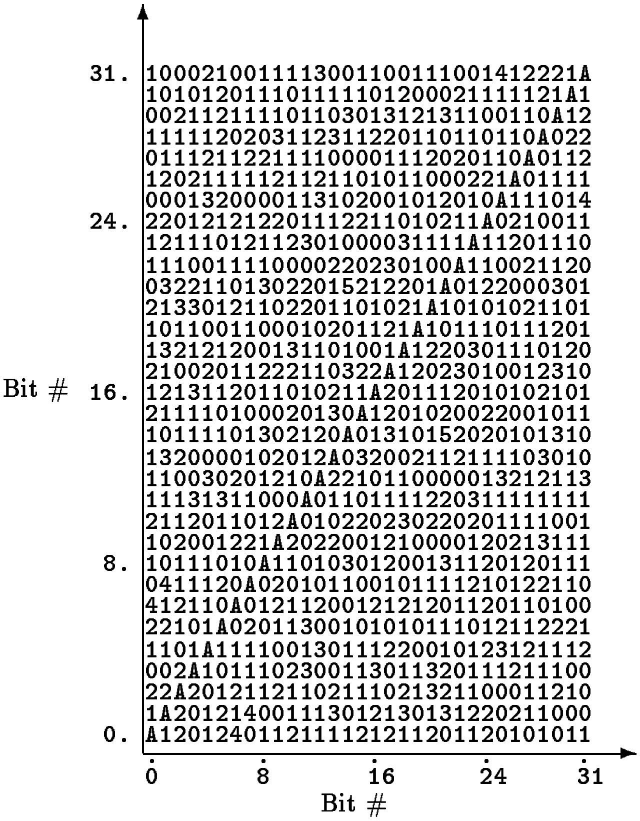


Figure 5: Information in CRC bits.



```
      31. 10002100111130011001110014122221A
          10101201110111110120002111112 1A1
```

Figure 6: Correlation among CRC bits.

1. Almost all 32 bits have a high information content very close to one bit. Thus, it does not matter which bit of the CRC we use.

2. All curves are (almost) horizontal straight lines. Thus, the information content of all bit combinations is identical. It does not matter which m bits are chosen for $m = 1, 2, \ldots$

3. The information content of m bits is approximately m. This means that CRC provides an almost optimal hashing function.

4. The curve for $m = 8$ is not so straight, but this is expected since we had only 296 distinct destination addresses and, thus, the total information content of the trace itself is $\log_2(296) = 8.21$ bits.

We also computed the correlation among CRC bits. The results are shown in Figure 6. As discussed earlier, only the first decimal digit of the correlation is shown. Notice that the correlation is very low and that occurrence of 'A' is limited to the diagonal. This once again confirms the conclusion that CRC is an excellent hashing function.

We repeated the analysis with several other 8-bit and 16-bit CRC polynomials. In general, we found that if a polynomial provides a good CRC, it can serve as an excellent hash function.

9 Hashing Using Fletcher Checksum

One problem with CRC is that its computation in software is complex (unless we use a tabular method, for example, that described in [24]). The ISO/OSI transport uses a checksum instead of CRC since it is so easy to compute in software. Although the checksum became widely known only after Fletcher proposed and analyzed it [5], it was discussed earlier by Samoylenko [26]. For a discussion on implementing it efficiently see [29]. Given an n-octet message b[1]...b[n], a two-octet checksum $C[0]$ and $C[1]$ is computed as follows:

```
C[0] = 0; C[1] =0;
For i = 1 step 1 until n do
C[0] = C[0] + b[i];
C[1] = C[1] + C[0];
EndFor;
```

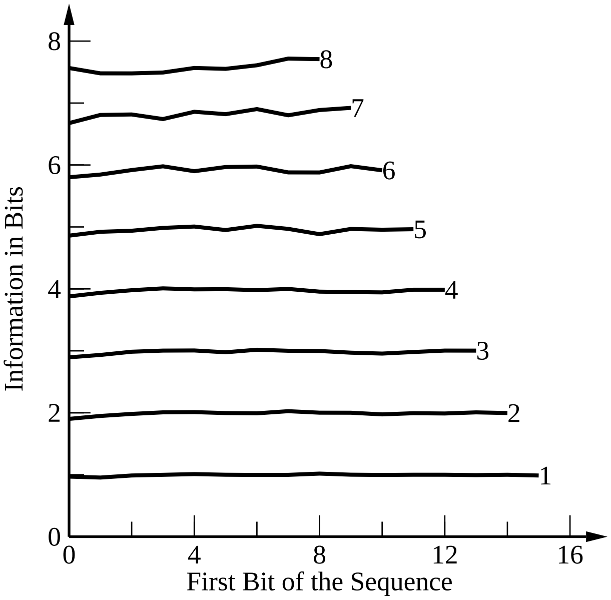


Figure 7: Information in Fletcher checksum bits.

Fletcher's procedure is more general than that described above in the sense that the checksum can be more than two octets long. However, OSI transport uses a two-octet checksum, which is what we analyze here. We computed the the information in m consecutive bits of address checksums. The results are shown in Figure 7.

From the figure we observe that each of the 16 bits individually have one bit of information and that the information content of m bits is approximately m. This is, therefore, also a good hashing function.

The correlation among the bits of the Fletcher checksum is shown in Figure 8. Notice that the correlation is small.

10 Hashing Using Another Checksum

Another popular checksum algorithm used to guard against memory errors in network address databases is [9]:

$$\begin{aligned} C &= Mod\,\big(2^8(4b[1] + 2b[3] + b[5]) + \\ &\quad (4b[2] + 2b[4] + b[6]), 2^16 - 1\big) \end{aligned}$$

Here, $b[i]$ is the ith octet of the address and C is the 16-bit checksum. Since we are not aware of its name, we will call it the 'mod-checksum.' The information content of the bits of this checksum are shown

```
            15. 320012111101102A
                21200121012202A2
                0143001001001A20
                110123132111A101
                00113022013A1021
                0114000210A31020
                231100113A011111
Bit #        8. 11101212A3102001
                1210031A21223011
                021100A111021121
                10111A0320003012
                2112A10010032001
                011A211001411300
                21A1111111110420
                2A11102213101112
             0. A220210112001023
                 .       .      .
                 0       8      15
                     Bit #
```

Figure 8: Correlation among bits of the Fletcher checksum.

in Figure 9. Comparing this figure with that for the Fletcher checksum, we find that the mod-checksum is not as good a hashing function as the Fletcher checksum even though it is more complex to compute. The key lesson is that *complexity does not necessarily get us better performance.*

The correlation among the bits of mod-checksum is shown in Figure 10.

11 Hashing Using XOR Folding

The final alternative that we investigated is that of the straightforward exclusive-or operation on the six octets of the address to produce 8 bits.

$$C = b[1] \oplus b[2] \oplus b[3] \oplus b[4] \oplus b[5] \oplus b[6]$$

The information content and correlation of the bits in the 'XOR-fold' so obtained are shown in Figures 11 and 12, respectively. To our surprise, this function, which is so simple to implement, is an excellent hashing function. In software, the folding may be preferable to Fletcher's checksum if exclusive-or operations take less time than additions. Implementing folding in hardware is simpler than CRC.

It should be pointed out that XOR-folding can be done with any number of bits. For example, a 11-bit fold can be obtained by dividing each 48-bit address in to five segments of which four are 11-bit long and

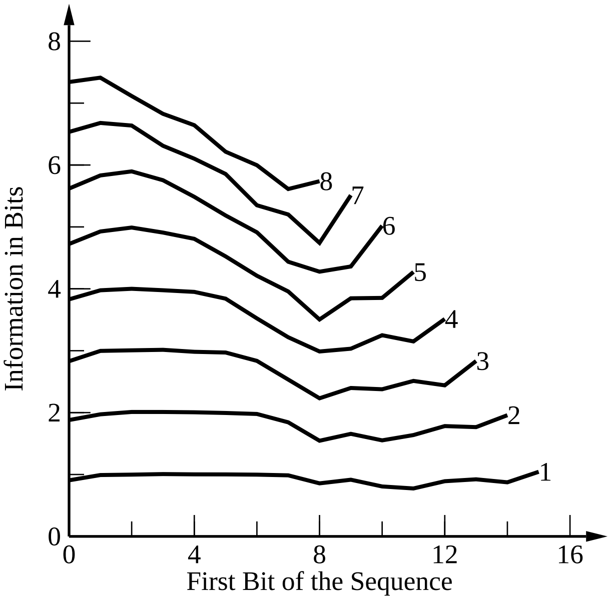


Figure 9: Information in mod-checksum bits.

```
            15. 201112022121101A
                42220121211132A1
                2101010122102A20
                021101105423A231
                02210112112A3011
                0111101044A22112
                131120216A414211
Bit #        8. 11111010A6415222
                2110202A01020112
                100201A212111020
                10010A1000011112
                1110A00212100001
                110A012011111121
                13A0100111121021
                1A31100113122120
             0. A111111211000242
                 .       .      .
                 0       8      15
                     Bit #
```

Figure 10: Correlation among bits of mod-checksum.

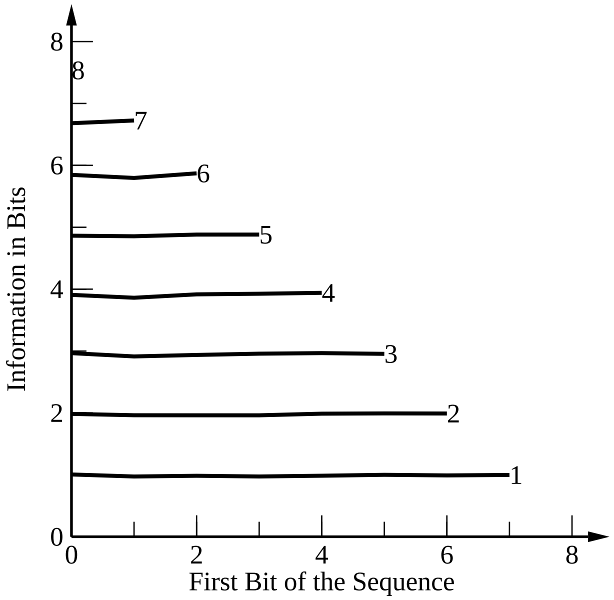
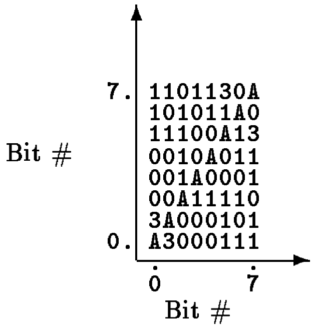


Figure 11: Information in XOR-fold bits.



```
1101130A
101011A0
11100A13
0010A011
001A0001
00A11110
3A000101
A3000111
```

Figure 12: Correlation among bits of XOR-fold.

the fifth one is 7-bit long. The exclusive-or of these five segments will produce a 11-bit fold.

12 Mask Size for an Address Filter

In this section, we address the problem of finding the size of the hash mask required to get a desired level of performance. We assume that the filter consists of a simple $M \times 1$ bit mask, that is, each hash table cell is only one bit wide. A hash function is used to map the address to an index value i in the range 0 through $M-1$, and if the ith bit in the mask is set, the frame is accepted for further processing; otherwise, the frame is rejected. This is how hashing is used in several commercial media access controller (MAC) chips. Such a hash filter is a perfect rejection filter in the sense that if the mask bit is zero, we can be certain that the address is not wanted, and there is no need to search the address table. On the other hand, such a hash filter is an imperfect acceptance filter in the sense that when the mask bit is one, there is some probability that the address is not one of those wanted by the station. This is because more than one addresses can hash to the same mask location.

The performance of the filter is measured by the probability of an unwanted address being rejected by the filter. We call this probability the **unwanted-rejection rate**. A larger mask will provide a better unwanted-rejection rate. For an infinitely large mask, the probability of an unwanted frame being rejected is 100%. For other size masks, the unwanted-rejection rate depends on several factors including the individual addresses that are wanted, their probability of being seen on the network, the addresses that are not wanted, and their probability of being seen on the network, etc. If we assume that all addresses are equally likely to be seen and that all mask cells are equally likely to be referred, then using the procedure described in Appendix A, we can compute the unwanted-rejection rate as a function of k – the number of addresses wanted, and M – the mask size. The final results are shown in Figure 13. In the figure, the number of addresses k that a station may want to receive is plotted along the horizontal axis, and the probability of rejecting an unwanted frame is plotted along the vertical axis. Eight curves corresponding to mask size M = 2, 4, 8, . . . , 128, 512 bits are shown.

From figure 13, we observe the following:

1. There is some filtering even with small masks. For example, if one wants to receive 10 addresses, an 8-bit mask is expected to reject 26% of the unwanted frames without further searching. Although this rate is low, the point is that it is non-zero even though the mask size is less than the number of addresses.

2. In general, it is better to have as large a mask as possible. For example, if one wants to receive 10 addresses with a 512-bit mask, 98% of the unwanted frames will be rejected without further searching.

3. If the mask size is large compared to the number of addresses to receive, that is, if $M \gg k$, the curves are linear and the unwanted-rejection rate is approximately $1 - k/M$.

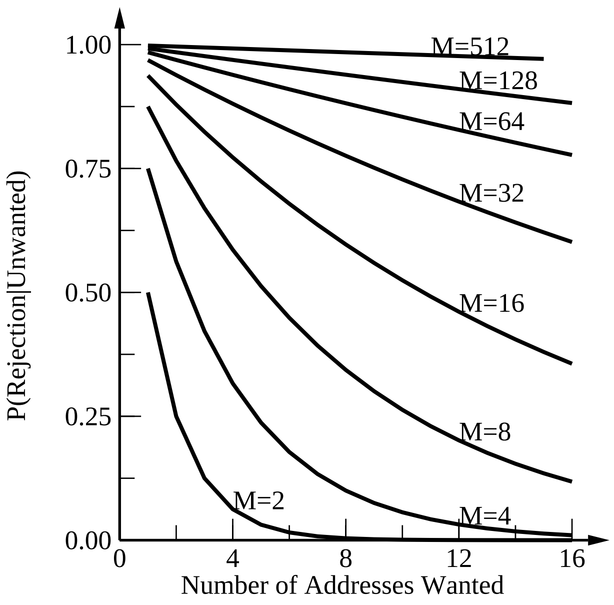


Figure 13: Probability of rejecting unwanted frames as a function of number of address wanted and the mask size M

The last observation is helpful in deciding the mask size. Thus, if one wants to reject 80% of the unwanted frames, the mask size should be 5 times the number of addresses desired.

The measurements on our extended LAN show that an average station currently listens to approximately 10 multicast addresses, i.e., $k \approx 10$. In the future, the number is expected to increase as more protocols with new multicast addresses are introduced. Thus, a good hash function with a 64-bit mask used to filter multicast addresses is currently expected to help avoid further searching for 85% of the unwanted frames.

13 SUMMARY

Since the size of computer networks continues to grow, we need to find ways to efficiently and quickly recognize destination addresses. In this paper, we investigated hashing as a possible solution to this problem. We showed that the number of lookups saved is equal to the information content of the bits of the hashed value.

We compared the performance of several hashing functions. First, simple bit extraction from the address itself provides a hashing function that is easy to implement in hardware as well as software. Second, bits extracted from the CRC of the address can be used as a hashing function that is easy to implement in hardware. Third, bits extracted from the Fletcher checksum can be used as a hashing function that is easy to implement in software. Finally, exclusive-or folding of the address octets provides another hashing alternative that is easy to implement both in software as well as hardware.

We concluded that CRC polynomials are excellent hashing functions. Fletcher's checksum and folding are also good hashing functions. The mod-checksum, which is more complex to compute than Fletcher's checksum, is not as good as the latter. Although bit extraction is not as good as other alternatives, it is the simplest. The choice between bit extraction and other alternatives is basically that of computing vs storage. If we can use excess memory, bit extraction with (a few more bits) may provide the same information as the checksum or folding with a few less bits.

It is interesting to find out which bits or octets of the address have high information. For example, for our trace we found that the fifth octet of the address had the highest information content. Therefore, this octet should be used first to hash addresses, to compare two addresses, to balance load among parallel routes, and as the branching function at the root of a tree structure.

We showed that for a station wanting to receive k addresses, the probability of rejecting unwanted frames using a simple $M \times 1$ bit mask is $1-k/M$. This allows us to decide the mask size required for a desired level of performance.

Some of the observations presented in this paper are limited to the environment that we measured. However, the methodology is general and can be applied to other environments and problems. In particular, it would be interesting to apply it to study the reference pattern of the 20-octet addresses used in the ISO network layer and the name reference patterns in various name servers and distributed systems.

14 Acknowledgments

We would like to thank Ruei-Hsin Hsiao of DEC for helping to gather the trace data. Thanks are also owed to Bill Hawe, Tony Lauck, Henry Yang, and Paul Koning of the Digital's networking architecture group for their valuable comments on earlier versions

of this paper.

References

[1] C. C. Chang, "The Study of an Ordered Minimal Perfect Hashing Scheme," Comm. of ACM, Vol. 27, No. 4, April 1984, pp. 384-387.

[2] R. J. Cichelli, "Minimal Perfect Hash Functions Made Simple," Comm. of ACM, Vol. 23, No. 1, January 1980, pp. 17-19.

[3] G. V. Cormack, R. N. S. Horspool, and M. Kaiserwerth, "Practical Perfect Hashing," The Computer Journal, Vol. 28, No. 1, 1985, pp. 54-58.

[4] R. J. Enbody and H. C. Du "Dynamic Hashing Schemes," ACM Computing Surveys, Vol. 20, No. 2, June 1988, pp. 85-112.

[5] J. G. Fletcher, "An Arithmetic Checksum for Serial Transmissions," IEEE Trans. on Communications, Vol. COM-30, No. 1, January 1982, pp. 247-252.

[6] J. Hamstra, "FDDI Design Tradeoffs," Proc. 13th Conf. on Local Computer Networks, Minneapolis, MN, October 10-12, 1988, pp. 297-300.

[7] W. Hawe, A. Kirby, and B. Stewart, "Transparent Interconnection of Local Networks with Bridges," Journal of Telecommunications Networks, Vol. 2, No. 2, September 1984, pp. 117-130.

[8] *Carrier Sense Multiple Access with Collision Detection (CSMA/CD)*, IEEE Standard 802.3-1985, 143 pp.

[9] *The Ethernet - A Local Area Network: Data Link Layer and Physical Layer Specifications*, Published jointly by Digital, Intel, and Xerox Corp, Version 2.0, November 1982, pp. 95-96.

[10] G. Jaeschke, "Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions," Comm. of ACM, Vol. 24, No. 12, December 1981, pp. 829-833.

[11] R. Jain and S. Routhier, "Packet Trains: Measurements and New Model for Computer Network Traffic," IEEE Journal on Special Areas in Communications, Vol. SAC-4, No. 6, September 1986, pp. 986-994.

[12] R. Jain, "Error Characteristics of FDDI," DEC Technical Report DEC-TR-553, June 1988, (Available from the author).

[13] R. Jain, "Characteristics of Destination Address Locality in Computer Networks: A Comparison of Caching Schemes," DEC Technical Report, DEC-TR-592, January 1989, (Available from the author).

[14] G. D. Knott, "Hashing Functions," The Computer Journal, Vol. 18, No. 3, August 1975, pp. 265-278.

[15] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp. 506-507.

[16] P. A. Larson, "Dynamic Hash Tables," Comm. of ACM, Vol. 31, No. 4, April 1988, pp. 446-457.

[17] T. G. Lewis and C. R. Cook, "Hashing for Dynamic and Static Internal Tables," IEEE Computer, October 1988, pp. 45-56.

[18] W. D. Maurer and T. G. Lewis, "Hash Table Methods," Computing Surveys, Vol. 7, No. 1, March 1975, pp. 5-19.

[19] R. Morris, "Scatter Storage Techniques," Comm. of ACM, Vol. 11, No. 1, January 1968, pp. 38-44. Reprinted in 25th anniversary issue of Comm. of ACM, Vol. 26, No. 1, January 1983, pp. 39-42.

[20] C. E. Price, "Table Lookup Techniques," Computing Surveys, Vol. 3, No. 2, June 1971, pp. 49-65.

[21] J. S. Quarterman and J. C. Hoskins, "Notable Computer Networks," Communications of the ACM, Vol. 29, No. 10, October 1986, pp. 932-971.

[22] F. E. Ross, "FDDI-A Tutorial," IEEE Communications Magazine, Vol. 24, No. 5, May 1986, pp. 10-17.

[23] T. J. Sager, "A Polynomial Time Generator for Minimal Perfect Hash Function," Comm. of ACM, Vol. 28, No. 5, May 1985, pp. 523-532.

[24] D. Sarvate, "Computation of Cyclic Redundancy Checks via Table Look-up," Comm. of ACM, Vol. 31, No. 8, August 1988, pp. 1008-1013.

[25] R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, 1988.

[26] S. I. Samoylenko, "Binoid Error-Correcting Codes," IEEE Trans. Inf. Theory, January 1973, pp. 95-101.

[27] D. G. Severance, "Identifier Search Mechanisms: A Survey and Generalized Model," Computing Surveys, Vol. 6, No. 3, September 1974, pp. 175-194.

[28] B. A. Sheil, "Median Split Trees: A Fast Lookup Technique for Frequently Occurring Keys," Comm. of ACM, Vol. 21, No. 11, November 1978, pp. 947-958.

[29] K. Sklower, "Improving the Efficiency of the ISO Checksum Calculation," Report No. UCB/CSD 88/427, September 1988, 9 pp.

[30] R. Sprugnoli, "Perfect Hashing Function: A Single Probe Retrieving Method for Static Sets," Comm. of ACM, Vol. 20, No. 11, November 1977, pp. 841-850.

[31] *Webster's Ninth New Collegiate Dictionary*, Merriam-Webster Inc., Springfield, MA, 1983.

15 Appendix A: Computation of the Unwanted-Rejection Rate

In this appendix, we derive an expression for the probability of rejecting unwanted frames using an $M \times 1$ bit hash mask and explain the procedure used to obtain Figure 13 presented earlier.

Let l be the number of bits set in an M-bit mask at a station that wants to receive k addresses. It is clear that l is less than or equal to k since a few addresses may collide, i.e., hash to the same cell. Let $p(l)$ be the probability that l bits in the mask are set. Since l must be between 1 and k, we have:

$$\sum_{l=1}^{k} p(l) \quad = \quad 1$$

If we assume that all addresses are equally likely to be seen and that all mask cells are equally likely to be referred, l/M of the frames will be accepted by the filter. For any given value of l, the probability of an unwanted address being accepted is l/M, that is:

$$P(\text{Rejection} \mid \text{Unwanted}, \, l) = 1 - \frac{l}{M}$$

Averaging over all values of l, we have:

$$\begin{aligned} P(\text{Rejection} \mid \text{Unwanted}) &= \sum_{l=1}^{k} p(l)\left(1 - \frac{l}{M}\right) \\ &= 1 - \frac{\sum_{l=1}^{k} l p(l)}{M} \\ &= 1 - \frac{\bar{l}}{M} \end{aligned}$$

Here, $\bar{l}$ is the mean value of l. If we hash k addresses on M mask cells, there are M^k possible combinations. Each of these combinations is equally likely, and if we know the number of combinations $N(k, l, M)$ that result in l distinct bits being set, we can compute $p(l)$ as follows:

$$p(l) = \frac{N(k, l, M)}{M^k}$$

For example, if we have a 4-bit mask ($M = 4$), and want to receive three addresses ($k = 3$), there are $4^3 = 64$ combinations as shown in Figure 14. Of these, four combinations involve only one distinct bit, 36 combinations involve two distinct bits, and 24 combinations involve three distinct bits. Thus, the probabilities $p(l)$ of l being 1, 2, or 3 are 4/64, 36/64, and 24/64, respectively. The mean value of l is:

$$\bar{l} = 1 \times \frac{4}{64} + 2 \times \frac{36}{64} + 3 \times \frac{24}{64} = \frac{148}{64} = 2.31$$

The expected probability of an unwanted address being rejected is:

$$P(\text{Rejection} \mid \text{Unwanted}) = 1 - \frac{2.31}{4} = 0.4225$$

Thus, a 4-bit mask is only 42% effective for a station that wants to receive three addresses.

The unwanted-rejection rate of a hash filter can be computed similarly for other values of k and M by enumerating all M^k combinations. For large values of M, the enumeration of all M^k combinations is rather cumbersome. Therefore, we have developed a recursive procedure to compute $N(k, l, M)$ as follows.

Imagine that we have k addresses which we want to hash on to M cells. We take the first address, it can hash to any one of the M cells. This results in M combinations denoted by $\{0, 1, 2, \ldots, M-1\}$. This is similar to that shown in Figure 14. We now hash the second address, this results into M^2 combinations denoted by $\{(0,0), (0,1), \ldots, (0, M-1), (0,1), (1,1), \ldots, (1, M-1), \ldots, (M-1, M-1)\}$ Some of these combinations, e.g., $(0,0), (1,1), \ldots$ involve only one distinct bit, while others involve two

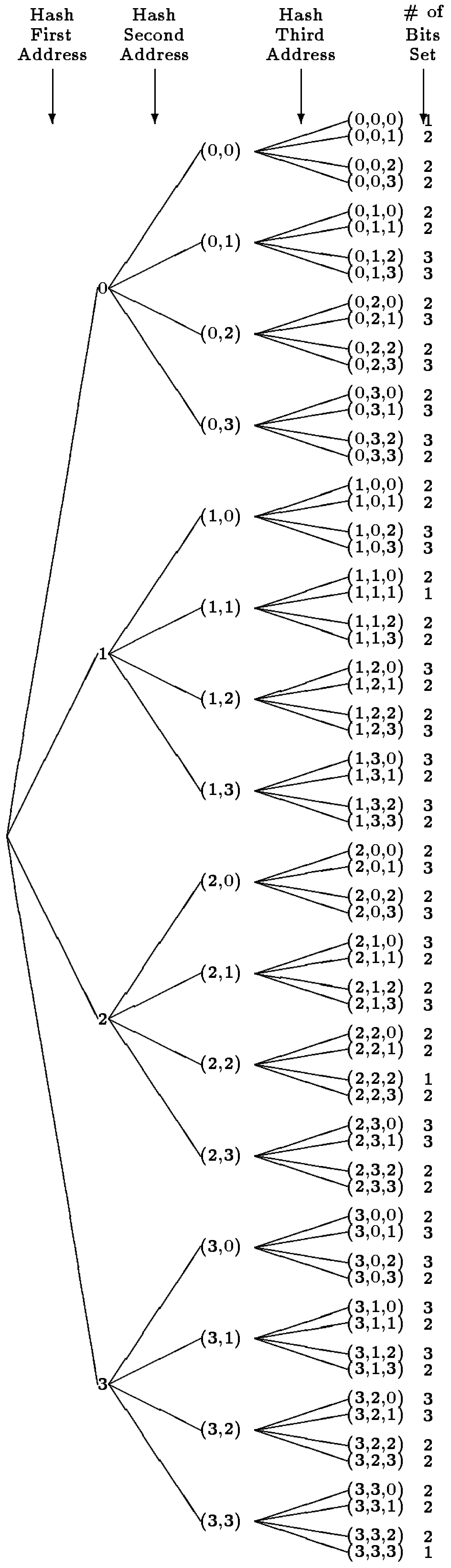


Figure 14: An example showing all possibilities if we hash three addresses on a four-bit mask.

distinct bits. We need to continue this process for k addresses. Consider the situation when we have completed hashing $k-1$ addresses. At this point, we know the number of combinations $N(k-1, l, M)$ involving l distinct bits for all values of l. At the next step, there are only two possibilities for each of the current M^{k-1} combinations: either the kth address hashes to one of the cells already occupied or it hashes to a cell not occupied. The first possibility results in l-bit combinations from existing l-bit combinations, the second possibility leads to l-bit combinations from existing $(l-1)$-bit combinations. This reasoning results in the following recursive formula for computing $N(k, l, M)$:

$$N(k,l,M) = lN(k-1,l,M)+(M-l+1)N(k-1,l-1,M) \tag{3}$$

The boundary conditions are:

$$\begin{aligned} N(0,0,M) &= 0 && \forall M \\ N(k,l,M) &= 0 && \forall l > k \end{aligned}$$

In general, $N(k, l, M)$ has the following form:

$$N(k,l,M) = g(k,l)M(M-1)\cdots(M-l+1) \tag{4}$$

Here, the coefficient $g(k, l)$ does not depend upon M. It can be precomputed and used for all values of M. Combining Equations 3 and 4, we get the following formula for computing $g(k, l)$:

$$g(k,l) = g(k-1,l-1) + lg(k-1,l) \tag{5}$$

The boundary conditions are:

$$\begin{aligned} g(k,1) &= 1 && \forall k \\ g(k,l) &= 1 && \forall l = k \end{aligned} \tag{6}$$

Equations 5 and 6 allow us to compute $g(k, l)$ in a recursive manner. The computed values of $g(k, l)$ for some values of k and l are listed in Table 2. Any entry in Table 2 can be computed from the two entries in the previous row – one directly above it and the second diagonally above to the left of it. Thus, this table can be easily extended row by row. For example, as shown in the table, $g(8, 4)$ can be computed as follows:

$$g(8,4) = g(7,3) + 4g(7,4) = 301 + 350 \times 4 = 1701$$

The following non-recursive formula for computing $g(k, l)$ can be obtained after some algebraic manipulation from Equations 5 and 6:

$$g(k,l) = \frac{1}{l!}\sum_{i=0}^{l-1}\binom{l}{i}(-1)^i(l-i)^k$$

Table 2: Coefficient $g(k,l)$

	l							
k	1	2	3	4	5	6	7	8
1	1							
2	1	1						
3	1	3	1					
4	1	7	6	1				
5	1	15	25	10	1			
6	1	31	90	65	15	1		
7	1	63	301	350	140	21	1	
				$\searrow$ $\downarrow \times 4$				
8	1	127	966	1701	1050	266	273	1

For example,

$$\begin{aligned} g(8,4) &= \frac{1}{4!}\left[\binom{4}{0}4^8 - \binom{4}{1}3^8 + \binom{4}{2}2^8 - \binom{4}{3}1^8\right] \\ &= \frac{65536-4\times 6561+6\times 256-4\times 1}{24} \\ &= \frac{40824}{24} = 1701 \end{aligned}$$

16 Appendix B: Numerical Results

In this paper, we have presented results graphically wherever possible. To allow easy reading of the values plotted, the same results are now presented in tabular form in this appendix.

Table 3: Information in Address Bits

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
0	0.4	1.4	1.4	1.4	1.4	1.4	1.4	1.4
1	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0
3	0.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0
4	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
5	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
6	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
10	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0
11	0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.3
12	0.0	0.0	0.0	0.0	1.0	1.0	1.3	1.3
13	0.0	0.0	0.0	1.0	1.0	1.3	1.3	1.3
14	0.0	0.0	1.0	1.0	1.3	1.3	1.3	1.3
15	0.0	1.0	1.0	1.3	1.3	1.3	1.3	1.3
16	1.0	1.0	1.3	1.3	1.3	1.3	1.3	1.3
17	1.0	1.3	1.3	1.3	1.3	1.3	1.3	1.7
18	1.0	1.2	1.2	1.2	1.2	1.3	1.6	1.9
19	1.0	1.0	1.0	1.0	1.0	1.5	1.8	1.9
20	0.0	1.0	1.0	1.0	1.5	1.8	1.9	1.9
21	1.0	1.0	1.0	1.5	1.8	1.9	1.9	1.9
22	0.0	0.0	0.5	1.4	1.6	1.6	1.6	1.6
23	0.0	0.5	1.4	1.6	1.6	1.6	1.6	1.6

Table 3: Information in Address Bits (Continued)

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
24	0.5	1.4	1.6	1.6	1.6	1.6	1.6	1.6
25	1.0	1.2	1.2	1.2	1.2	1.2	1.2	2.2
26	0.2	0.2	0.2	0.2	0.2	0.2	1.2	2.2
27	0.0	0.0	0.0	0.0	0.0	1.0	2.0	3.0
28	0.0	0.0	0.0	0.0	1.0	2.0	3.0	4.0
29	0.0	0.0	0.0	1.0	2.0	3.0	4.0	4.9
30	0.0	0.0	1.0	2.0	3.0	4.0	4.9	5.8
31	0.0	1.0	2.0	3.0	4.0	4.8	5.8	6.6
32	1.0	2.0	3.0	4.0	4.8	5.8	6.6	7.3
33	1.0	2.0	3.0	3.9	4.8	5.8	6.5	7.3
34	1.0	2.0	3.0	3.9	4.9	5.7	6.6	7.2
35	1.0	2.0	3.0	4.0	4.8	5.8	6.4	7.0
36	1.0	2.0	3.0	3.8	4.8	5.6	6.3	6.7
37	1.0	2.0	2.8	3.8	4.6	5.4	5.9	6.4
38	1.0	1.8	2.8	3.6	4.4	5.0	5.7	5.9
39	0.8	1.8	2.7	3.6	4.2	4.9	5.3	5.6
40	1.0	1.8	2.7	3.3	4.0	4.4	4.8	5.1
41	0.8	1.7	2.4	3.0	3.5	4.0	4.3	
42	0.9	1.6	2.3	2.9	3.5	3.9		
43	0.9	1.6	2.3	3.0	3.5			
44	0.8	1.5	2.3	2.9				
45	0.8	1.7	2.2					
46	0.9	1.6						
47	0.9							

Table 4: Information in CRC Bits

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
0	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.7
1	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.8
2	1.0	2.0	3.0	4.0	5.0	6.0	7.0	7.7
3	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.7
4	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.7
5	1.0	2.0	3.0	4.0	5.0	5.9	6.8	7.4
6	1.0	2.0	3.0	4.0	5.0	6.0	6.6	7.4
7	1.0	2.0	3.0	4.0	5.0	5.7	6.6	7.4
8	1.0	2.0	3.0	4.0	5.0	6.0	6.8	7.6
9	1.0	2.0	3.0	4.0	4.9	5.9	6.7	7.1
10	1.0	2.0	3.0	4.0	4.9	6.0	6.5	7.2
11	1.0	2.0	3.0	4.0	5.0	5.9	6.7	7.6
12	1.0	2.0	3.0	4.0	4.9	5.9	6.9	7.7
13	1.0	2.0	3.0	4.0	4.9	5.9	6.9	7.7
14	1.0	2.0	3.0	4.0	5.0	5.9	6.8	7.5
15	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.7
16	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.5
17	1.0	2.0	3.0	4.0	5.0	6.0	6.8	7.6
18	1.0	2.0	3.0	4.0	5.0	5.9	6.8	7.6
19	1.0	2.0	3.0	4.0	5.0	5.9	6.8	7.7
20	1.0	2.0	3.0	4.0	4.9	5.9	6.8	7.7
21	1.0	2.0	3.0	4.0	4.9	5.8	6.8	7.5
22	1.0	2.0	3.0	3.9	4.9	5.8	6.7	7.5
23	1.0	2.0	3.0	3.9	5.0	5.9	6.8	7.6
24	1.0	2.0	3.0	4.0	4.9	5.9	6.8	7.6
25	1.0	2.0	3.0	3.9	4.9	5.9	6.7	
26	1.0	2.0	3.0	4.0	4.9	5.9		
27	1.0	2.0	3.0	4.0	4.9			
28	1.0	2.0	3.0	4.0				
29	1.0	2.0	3.0					
30	1.0	2.0						
31	1.0							

Table 5: Information in Fletcher Checksum Bits

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
0	1.0	1.9	2.9	3.9	4.9	5.8	6.7	7.6
1	1.0	1.9	2.9	3.9	4.9	5.8	6.8	7.5
2	1.0	2.0	3.0	4.0	4.9	5.9	6.8	7.5
3	1.0	2.0	3.0	4.0	5.0	6.0	6.7	7.5
4	1.0	2.0	3.0	4.0	5.0	5.9	6.9	7.6
5	1.0	2.0	3.0	4.0	5.0	6.0	6.8	7.6
6	1.0	2.0	3.0	4.0	5.0	6.0	6.9	7.6
7	1.0	2.0	3.0	4.0	5.0	5.9	6.8	7.7
8	1.0	2.0	3.0	4.0	4.9	5.9	6.9	7.7
9	1.0	2.0	3.0	3.9	5.0	6.0	6.9	
10	1.0	2.0	3.0	3.9	5.0	5.9		
11	1.0	2.0	3.0	4.0	5.0			
12	1.0	2.0	3.0	4.0				
13	1.0	2.0	3.0					
14	1.0	2.0						
15	1.0							

Table 7: Information in XOR-Fold Bits

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
0	1.0	2.0	3.0	3.9	4.9	5.8	6.7	7.5
1	1.0	2.0	2.9	3.9	4.9	5.8	6.7	
2	1.0	2.0	2.9	3.9	4.9	5.9		
3	1.0	2.0	3.0	3.9	4.9			
4	1.0	2.0	3.0	3.9				
5	1.0	2.0	3.0					
6	1.0	2.0						
7	1.0							

Table 6: Information in Mod-Checksum Bits

Starting	# of Bits							
at Bit	1	2	3	4	5	6	7	8
0	0.9	1.9	2.8	3.8	4.7	5.6	6.5	7.3
1	1.0	2.0	3.0	4.0	4.9	5.8	6.7	7.4
2	1.0	2.0	3.0	4.0	5.0	5.9	6.6	7.1
3	1.0	2.0	3.0	4.0	4.9	5.8	6.3	6.8
4	1.0	2.0	3.0	4.0	4.8	5.5	6.1	6.6
5	1.0	2.0	3.0	3.8	4.5	5.2	5.9	6.2
6	1.0	2.0	2.8	3.5	4.2	4.9	5.3	6.0
7	1.0	1.8	2.5	3.2	4.0	4.4	5.2	5.6
8	0.9	1.5	2.2	3.0	3.5	4.3	4.7	5.7
9	0.9	1.7	2.4	3.0	3.8	4.4	5.5	
10	0.8	1.6	2.4	3.3	3.9	5.0		
11	0.8	1.6	2.5	3.1	4.3			
12	0.9	1.8	2.4	3.5				
13	0.9	1.8	2.8					
14	0.9	2.0						
15	1.0							

Table 8: Percent Unwanted-Rejection Rate

	Mask Size M							
k	2	4	8	16	32	64	128	512
1	50.0	75.0	87.5	93.8	96.9	98.4	99.2	99.8
2	25.0	56.3	76.6	87.9	93.8	96.9	98.4	99.6
3	12.5	42.2	67.0	82.4	90.9	95.4	97.7	99.4
4	6.3	31.6	58.6	77.2	88.1	93.9	96.9	99.2
5	3.1	23.7	51.3	72.4	85.3	92.4	96.2	99.0
6	1.6	17.8	44.9	67.9	82.7	91.0	95.4	98.8
7	0.8	13.3	39.3	63.7	80.1	89.6	94.7	98.6
8	0.4	10.0	34.4	59.7	77.6	88.2	93.9	98.4
9	0.2	7.5	30.1	55.9	75.1	86.8	93.2	98.3
10	0.1	5.6	26.3	52.4	72.8	85.4	92.5	98.1
11	0.0	4.2	23.0	49.2	70.5	84.1	91.7	97.9
12	0.0	3.2	20.1	46.1	68.3	82.8	91.0	97.7
13	0.0	2.4	17.6	43.2	66.2	81.5	90.3	97.5
14	0.0	1.8	15.4	40.5	64.1	80.2	89.6	97.3
15	0.0	1.3	13.5	38.0	62.1	79.0	88.9	97.1