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

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.

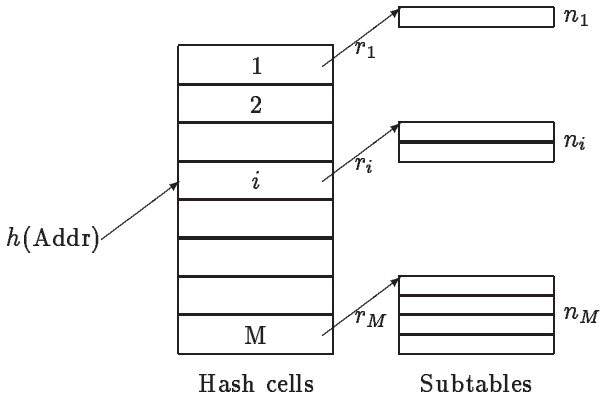

Figure 1: Hashing concepts.

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 total 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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.