II. A Scatter Storage Scheme For Dictionary Lookups

D. M. Murray

1. Introduction

A document retrieval system must have some means of recording the subject matter of each document in its data base. Some systems store the actual text words, while others store keywords or similar content indicators. The SMART system [1] uses concept numbers for this purpose, each number indicating that a certain word appears in the document. Two advantages are apparent. First, a concept number can be held in a fixed sized storage element. This produces faster processing than if variable sized keywords were used. Second, the amount of storage required to hold a concept number is less than that needed for most text words. Hence, storage space is used more efficiently.

SMART must be able to find the concept numbers for the words in any document or query. This is done by a dictionary lookup. There are two reasons why the lookup must be rapid. For text lookups, a slow scheme is costly because of the large number of words to be processed. For handling user queries in an on-line system, a slow lookup adds to the user response time.

Storage space is also an important consideration. Even for moderate sized subject areas the dictionary can become quite large — too large for computer main memory, or so large that the operation of the rest of the retrieval system is penalized. In most cases a certain amount of core storage is allotted to the dictionary, and the lookup scheme must do the best possible job within this allotment. This usually means keeping the overhead for the scheme as low as possible so that a large portion of the allotted core is available to hold dictionary words. The rest of the dictionary is placed in auxiliary storage and parts of it are brought in as needed. Obviously the number of accesses to auxiliary storage must be minimized.

This paper represents a study of scatter storage schemes for application to dictionary lookup. These methods appear to be fast and yet conservative with storage. The next two sections describe scatter storage schemes in general. The fourth section presents the results of various experiments with hash coding algorithms. The fifth section discusses the design and use of a practical lookup scheme. The final sections deal with extensions and conclusions.

2. Basic Scatter Storage

A) Method

A basic scatter storage scheme consists of a transformation algorithm and a table. The table serves as the dictionary and is constructed as follows. Given a natural language word, the algorithm operates on its bit pattern to produce an address, and the concept number for the word is placed in the table slot indicated by this address. This process is repeated for every word to be placed in the dictionary. The generated addresses are called hash addresses; and the table, a hash table.

There are many possible algorithms for producing hash addresses. [2,3,4] Some of the most common are:

1) choosing bits from the square of the integer represented by

the input word;

2) cutting the bit pattern into pieces and adding these pieces;

3) dividing the integer represented by the input word by the length of the hash table and using the remainder.

B) Collisions

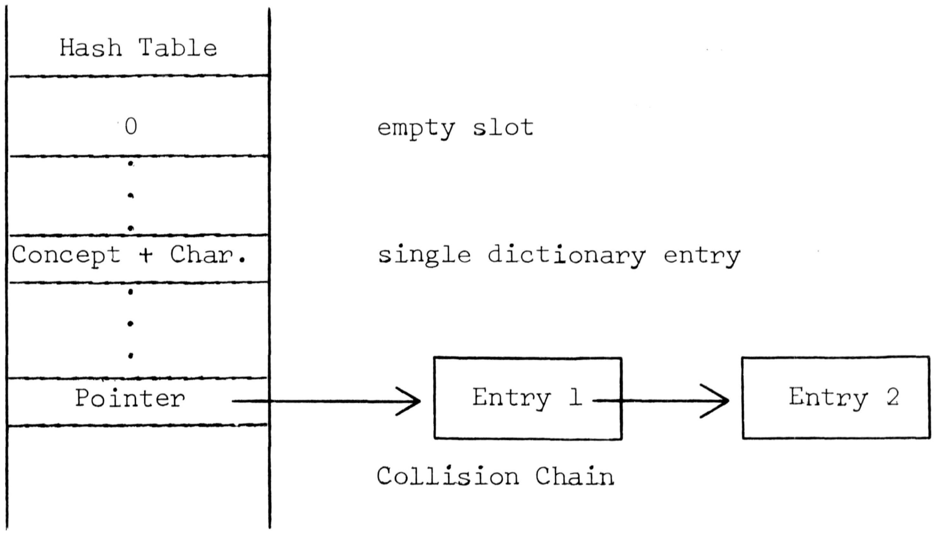
The ideal situation would arise if every word placed in the dictionary had a unique hash address. However, as soon as a few slots in the hash table have been filled, the possibility of a collision arises — two or more words producing the same hash address. To differentiate among collided entries, the characters of the dictionary words must be stored along with their concept numbers. Then during lookup, the input word can be compared with the character string to verify that the correct table entry has been located.

The problem of where to store the collided items has several methods of solution. [3] The linear scan method places a collided item in the first free table slot after the slot indicated by the hash address. The scan is circular over the end of the table. The random probe method uses a crude algorithm to generate random offsets $R(i)$ in the interval $[1,H]$ where H is the length of the hash table. If the colliding address is A, slot $A+R(1) \bmod H$ is examined. The process is repeated until an empty slot is found. Both of these methods work best when the hash table is lightly loaded, that is when the ratio between the number of words entered and the number of table slots is small. In such cases the expected length of scan or average number of random probes is small.

Chaining methods provide a satisfactory method of resolving collisions regardless of the load on the hash table. However, they require a second storage table — a bump table — for holding the collided items. When a collision occurs, both entries are linked together by a pointer and placed in the bump table. A pointer to this collision chain in placed in the hash table along with an identifying flag. Further colliding items are simply added to the end of the collision chain.

C) Table Layout and Search Procedure

In the virtual scatter storage system described later, the hash table has a high load factor. Hence the chained method (or rather a variation of it) is used to resolve collisions. Further discussion involves only scatter storage systems using collision chains. With this restriction, then, a scatter storage system consists of a hash table, a bump table, and the associated algorithm for producing hash addresses. A dictionary entry consists of a concept number and the character string for the word it represents. These entries are placed in the hash-bump table as described above. Consequently there are three types of slots in the hash table — slots that are empty, slots holding a single dictionary entry, and slots containing a pointer to a collision chain held in the bump table. A typical table layout is shown below.



One of the advantages of scatter storage systems is that the search strategy is the same as the strategy for constructing the hash-bump tables. Given a word, its hash address is computed and the tables searched to find the proper slot. During construction, dictionary information is placed in the slot. The basic search procedure is illustrated by the flow diagram in Fig. 1. The construction procedure is similar.

D) Theoretical Expectations

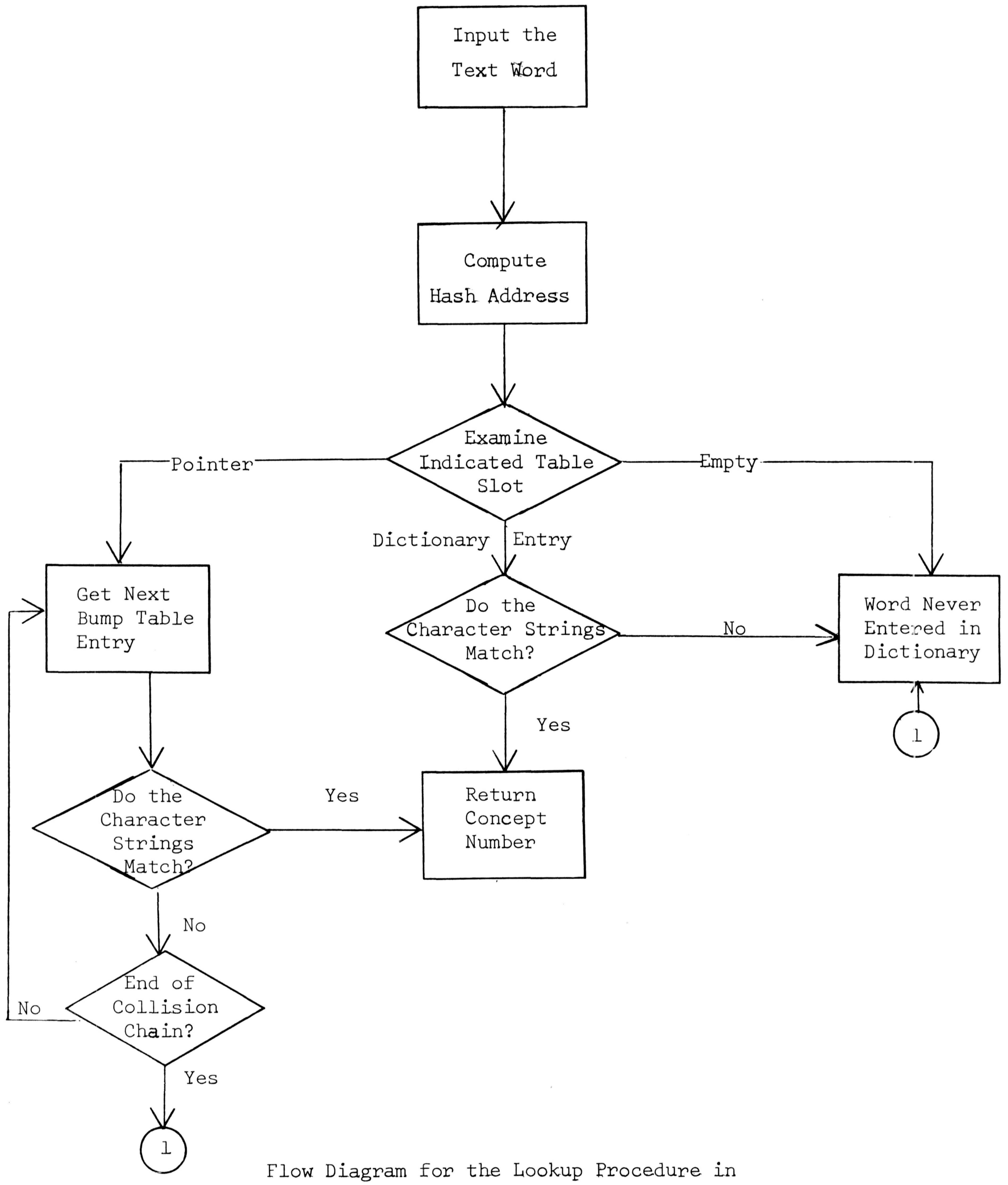
An ideal transformation algorithm produces a unique hash address for each dictionary word and thereby eliminates collisions. From a practical point of view, the best algorithms are those which spread their addresses uniformly over the table space. Producing a hash address is simply the process of generating a uniform random number from a given character string. If the addresses are truly random, a probability model may be used to predict various facts about the storage system.

Suppose a hash table has H slots and that N words are to be entered in the hash-bump tables. Let H_i be the expected number of hash table slots with i entries for $i = 0,1,...N$. In other words H_0 is the expected number of empty slots, H_1 is the expected number of single entries, and $H_2,H_3,...,H_N$ are the expected number of slots with various numbers of colliding items. Even though the items are physically located in the bump table, they may be considered to "belong" to the same slot in the hash table.

It is expected that:

$$(1)\quad H = \sum_{i=0}^{N} H_i$$

$$(2)\quad N = \sum_{i=0}^{N} i\, H_i$$


Input the Text Word
Compute Hash Address
Examine Indicated Table Slot
Pointer
Empty
Dictionary Entry
Get Next Bump Table Entry
Do the Character Strings Match?
No
Word Never Entered in Dictionary
Yes
1
Do the Character Strings Match?
Yes
Return Concept Number
No
End of Collision Chain?
No
Yes
1

Flow Diagram for the Lookup Procedure in
Basic Scatter Storage Systems

Fig. 1

Now let

$$X_{ij} = \begin{cases} 1 \text{ if exactly } i \text{ items occur in the } j^{th} \text{ slot} \\ 0 \text{ if exactly } i \text{ items do not occur in the } j^{th} \text{ slot} \end{cases}$$

for $j = 1, 2, . . . , H$

Then $H_i = E\{X_{i1} + X_{i2} + . . . + X_{iH}\}$

$$= \sum_{j=1}^{H} E\{X_{ij}\}$$

Assume that any chosen table slot is independent of the others so that the probability of getting any single item in the slot is $1/H$. Then the probability of getting exactly i items in that slot is

$$(3) \quad P_i = \binom{N}{i}\left(\frac{1}{H}\right)^i \left(1 - \frac{1}{H}\right)^{N-i}$$

Then $$E\{X_{ij}\} = 1\cdot P_i + 0\cdot(1-P_i)$$
$$= P_i$$

Substituting into the above

$$(4) \quad H_i = H\cdot P_i$$
$$= H\binom{N}{i}\left(\frac{1}{H}\right)^i \left(1 - \frac{1}{H}\right)^{N-i} \quad \text{for } i = 0,1, . . . , N$$

For the cases of interest H and N are large, and the Poisson approximation can be used in equation (3):

$$P_i = e^{-N/H} \frac{(N/H)^i}{i!}$$

The ratio N/H is the load factor mentioned previously. It is usually designated by α so that

$$(5)\quad H_i = H e^{-\alpha} \frac{\alpha^i}{i!} \qquad i = 0,1, \ldots, N$$

In a form more convenient for hand computation

$$H_o = e^{-\alpha}$$

$$H_i = H_{i-1}\frac{\alpha}{i} \qquad i = 1,2, \ldots, N$$

Equation (5) is sufficient to describe the state of the scatter storage system after the entry of N items. Most of the statistics of interest can be preducted using this expression; a few of them are listed in Table 1.

The time required for a single lookup using a hash scheme depends on the number of probes into the table space, that is, how many slots must be examined. Suppose the word is actually found. If it is a single entry, only one probe is required. If the word is located in a collision chain, the number of probes is one (for the hash table) plus one additional probe for each element of the collision chain that must be examined. Suppose that the word is not in the dictionary. If its hash address corresponds to an empty table slot, again only one probe is needed. However, if the address points to a collision chain, the number is one plus the length of the chain.

For words found in the dictionary the average number of probes per lookup is:

$$(8)\quad P = 1 + \frac{1}{N}\{(0)H_1 + (1+2)H_2 + (1+2+3)H_3 + \ldots + (1+2+\ldots+N)H_N\}$$

$$= 1 + \sum_{i=2}^{N} H_i \sum_{j=1}^{i} j$$

Measure	Formula
Load factor	$\alpha = N/H$
Number of empty table slots	$H_o = H\,e^{-\alpha}$
Number of single entries	$H_1 = N\,e^{-\alpha}$
Number of collision chains of length i	$H_i = H\,e^{-\alpha}\frac{\alpha^i}{i!} \quad i = 2,3,\ldots,N$
Expected sums	$H = \sum_{i=o}^{N} H_i$
	$N = \sum_{i=o}^{N} iH_i$
Fraction of hash table empty	$F_o = \frac{1}{H} H_o = e^{-\alpha}$
Fraction of table filled with single entries	$F_1 = \frac{1}{H} H_1 = \alpha e^{-\alpha}$
Fraction of hash table slots with i entries	$F_i = \frac{1}{H} H_i = e^{-\alpha}\frac{\alpha^i}{i!} \quad i=2,3,\ldots,N$
Expected sums	$1 = \sum_{i=o}^{N} F_i$
	$\alpha = \sum_{i=o}^{N} i\,F_i$
Number of collisions	$N_c = H_2 + H_3 + \ldots + H_N$ $= H - H_o - H_1$
Number of entries in the bump table	$B = N - H_1$
Total table slots required	$S = M + B$
Average lookup time (probes)	$P = 2 + \frac{\alpha}{2} - e^{-\alpha}$

H = number of hash table slots
N = number of words to be entered

Expected Storage and Search Properties for
Basic Scatter Storage Schemes

Table 1

$$= 1 + \frac{H}{N}\sum_{i=2}^{N} \frac{H_i}{H} \frac{i(i+1)}{2}$$

$$= 1 + \frac{1}{2\alpha} \sum_{i=2}^{N} i(i+1)F_i$$

$$= 1 + \frac{1}{2} \sum_{i=2}^{N} (i+1)F_{i-1}$$

$$= 1 + \frac{1}{2} \sum_{i=2}^{N} (i-1)\, F_{i-1} + \frac{1}{2} \sum_{i=2}^{N} F_{i-1}$$

$$\doteq 1 + \frac{1}{2} \sum_{i=2}^{N+1} (i-1)F_{i-1} + \frac{1}{2} \sum_{i=2}^{N+1} F_{i-1}$$

$$= 1 + \frac{1}{2}\alpha + (1-F_o)$$

$$= 2 + \frac{1}{2}\alpha - e^{-\alpha} \qquad \text{(probes)}$$

3. Virtual Scatter Storage

A) Method

From Table 1, the expected number of collisions is

$$N_c = H - H_o - H_1$$

$$= H(1 - e^{-N/H} - \frac{N}{H} e^{-N/H})$$

For a fixed N, this number decreases as H increases. At the same time the number of empty hash table slots

$$H_o = H\, e^{-N/H}$$

increases as H increases. Both of these results are expected — as the hash addresses are spread over a larger and larger table space (H slots), the number of collisions should decrease and the number of empties increase for a fixed number of entries (N).

A virtual scatter storage scheme tries to balance these opposing strains by combining hash coding with a sparse storage technique. Large or virtual hash addresses are used to obtain the collision properties associated with a very large hash table, and the storage technique is used to achieve the storage and search properties of a reasonably sized hash table. If the virtual hash address is taken large enough the expected number of collisions can be reduced to essentially zero. With no expected collisions, it is possible to dispense with verifying that a query word and the dictionary word are the same. It is enough to check that they produce the same virtual address. Hence, the character strings need not be stored in the hash-bump tables at all.

To implement the virtual scheme a large hash address is computed, say in the range [0,V], and is split into a major and minor part. The major portion is used just as before — as an index on a hash table of size H. Instead of storing the character string in the hash or bump table, the minor portion is substituted. With this difference, the virtual scheme works just as the basic scheme. The lookup procedure is identical, but the minor portions are used for comparison rather than character strings. All the results of the previous section apply as storage and timing estimates.

The advantage of virtual scatter storage systems is economy of storage space. The size of the minor portion is much smaller than the size of the character string it replaces. It is true, that the virtual scheme assigns the same concept number to two different words if they have the same virtual address. This need not be disastrous for document retrieval applications. Presumably V is chosen large enough to keep the number of collisions small. On the one hand, errors could be neglected because of their low probability of occurrance and their small effect on the total performance of the retrieval system. On the other hand, it is always possible to resolve detected collisions even in a virtual scheme. Collisions may be detected during dictionary construction or updating, and the characters for the colliding words appended to the bump table. The hash or bump table entry must contain a pointer to these characters along with an identifying flag. Collisions occurring during actual lookups cannot be detected.

B) Collision Problem

In order to use a virtual hash scheme, the virtual table must be large enough to reduce to expected number of collisions to an acceptable level. From a practical point of view, a collision may be considered to involve only 2 words, rather than 3, 4, or more. It is assumed that the probability of these other types of collisions is negligible. Let V be the size of the virtual hash table. Then the expected number of collisions is simply

$$\begin{aligned} N_c &= H_2 \\ &= V \frac{\alpha^2}{2} e^{-\alpha} \end{aligned}$$

where $\alpha = \frac{N}{V}$. In this case $V >> N$ so that α is small and $e^{-\alpha}$ is approximately 1.

$$(9)\quad N_c = V\frac{\alpha^2}{2}$$

$$= \frac{N^2}{2V}$$

Suppose, for example, the dictionary has $N = 2^{13}$ words. If the size of the virtual hash table is chosen to be $V = 2^{26}$, then the expected number of collisions is

$$N_c = \frac{(2^{13})^2}{2(2^{26})} = \frac{1}{2}$$

Suppose further that this table size is adopted for the dictionary, and that the hash code algorithm produces 3 collisions. The question arises whether the algorithm is a good one — whether it produces uniform random addresses. The answer is found by extending the previous probability model.

Consider a virtual scatter storage scheme in which the virtual table size is V, and N items are to be entered into the hash-bump tables. Again assume that collisions involve only two items. Let

$$P(i) = \text{Prob}\,\{i \text{ collisions}\}$$

$$= \text{Prob}\,\{i \text{ table slots have 2 items and } N-2i \text{ slots have 1 item}\}$$

The number of ways of choosing the i pairs of colliding words (in an ordered way) is:

$$\binom{N}{2}\binom{N-2}{2}\cdots\binom{N-2i+2}{2} = \frac{N!}{2^i\,(N-2i)!}$$

There are $i!$ ways of ordering these pairs and

$$(V)_{N-i} = \frac{V!}{(V-N+i)!}$$

ways of placing the pairs in the hash table, so that

$$(10) \quad P(i) = \frac{N!}{2^i i! \; (N-2i)!} \quad \frac{(V)_{N-i}}{V^N} \qquad \text{for } i = 0,1,\ldots,\lfloor \tfrac{N}{2} \rfloor$$

In a form for hand computation,

$$(11) \quad P(0) = (1 - \tfrac{1}{V})\,(1 - \tfrac{2}{V})\;.\;.\;.\;(1 - \tfrac{N-1}{V})$$

$$P(i) = P(i-1)\,\frac{(N-2i+2)\,(N-2i+1)}{2i(V-N+i)} \qquad \text{for } i = 1,2,\ldots,\lfloor \tfrac{N}{2} \rfloor$$

These results are exact, but the following approximations can be used with accuracy

$$\log P(0) = \sum_{j=1}^{N-1} \log\,(1 - \tfrac{j}{V})$$

$$= \sum_{j=1}^{N-1} - \frac{j}{V}$$

$$= - \frac{N^2}{2V}$$

Let $\beta = \frac{N^2}{2V}$. Terms linear in N may be neglected in equation (11) giving

$$P(0) = e^{-\beta}$$

$$P(i) = \frac{\beta}{i} P(i-1)$$

This is also a Poisson distribution:

$$(12) \quad P(i) = e^{-\beta} \frac{\beta^i}{i!} \qquad \text{for } i=0,1,2,\ldots,\lfloor \tfrac{N}{2} \rfloor$$

This equation gives the approximate probability of i collisions for a virtual scatter storage scheme. It may be used to form a confidence interval around the expected number of collisions $N_c = \beta$.

For the previous example in which $V = 2^{26}$, $N = 2^{13}$, $N_c = \frac{1}{2}$, the following table of values can be made:

i	$P(i)$	$\sum P(i)$
0	.607	.607
1	.303	.910
2	.076	.986
3	.012	.998

The probability is .986 that the number of collisions is less than or equal to 2. Since the algorithm gave 3 collisions, it appears to be a poor one. The results for the collision properties are summarized in Table 2.

4. Experiments with Algorithms for Generating Hash Addresses

Any scatter storage scheme depends on a good algorithm for producing hash addresses. This especially is true for virtual schemes in which collisions are to be eliminated. In these experiments three basic algorithms are evaluated for use in virtual schemes. The words in two dictionaries — the ADI Wordform and CRAN 1400 Wordform — are used. The hash-bump tables are filled using these words and the resulting collision and storage statistics compared with the expected values.

A) Dictionaries

The ADI Wordform contains 7822 words pertaining to the field of docu-

Measure	Formula
Collision factor	$\beta = \frac{N^2}{2V}$
Expected number of collisions	$N_c = \beta$
Probability of i collisions	$P(i) = e^{-\beta} \frac{\beta^i}{i!}$ $i=0.\ 1,\ldots,$
Probability that the number of collisions C lies in [a,b]	$\text{Prob} = \sum_{i=a}^{b} P(i)$

V virtual hash table size

N number of words to be entered

Expected Collision Properties for
Virtual Scatter Storage Systems

Table 2

mentation. It contains 206 common words (previously judged) averaging 3.93 characters. The remaining 7616 noncommon words average 8.00 characters. In all there are 61,712 characters.

The CRAN 1400 Wordform contains 8926 words dealing with aeronautics. The common word list consists of that of the ADI, plus four additional entries. The 8716 noncommon words average 8.40 characters. There is a total of 74,074 characters.

Figs. 2 and 3 show the distribution of the length of the words versus percentage of collection. The abrupt end to the curves in Fig. 2 is due to truncation of words to 18 characters.

Both dictionaries have approximately the same size and proportions of words of various length. However, their vocabularies are considerably different. A good hash scheme should work equally well on both dictionaries.

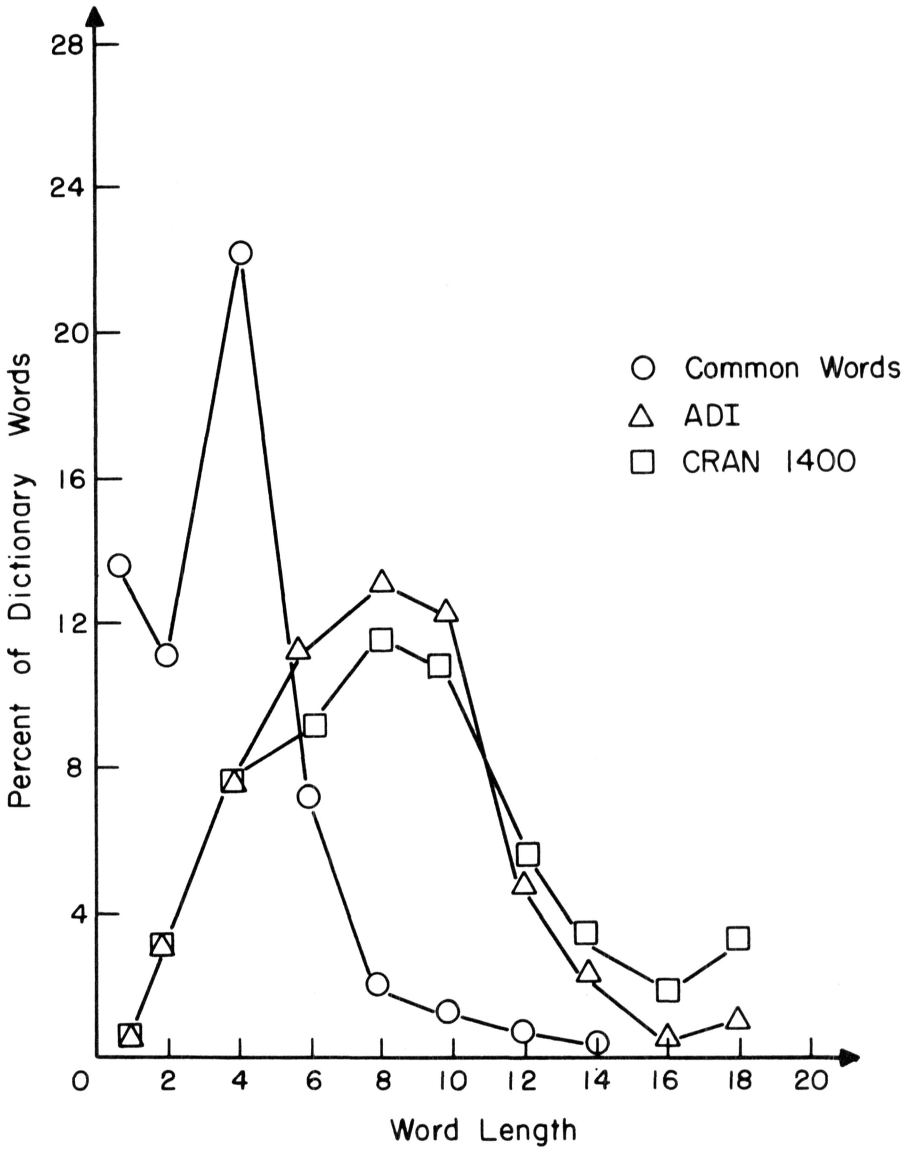
B) Hash Coding Algorithms

By their nature, hash coding algorithms are machine dependent. The computer representation of the alphabetic characters, the way in which arithmetic operations are done, and other factors all affect the randomness of the generated address. The algorithms described below are intended for use on the IBM S/360.

Words are padded with some character to fill an integral number of S/360 fullwords. Then the fullwords are combined in some manner to form a single fullword key, and the final hash address is computed from this key. In the experiments which follow, the blank is used as a fill character. This is an unfortunate choice because of the binary representation of the blank 01000000. In some algorithms the zeroes may propagate or otherwise affect the randomness. A good fill character is one that

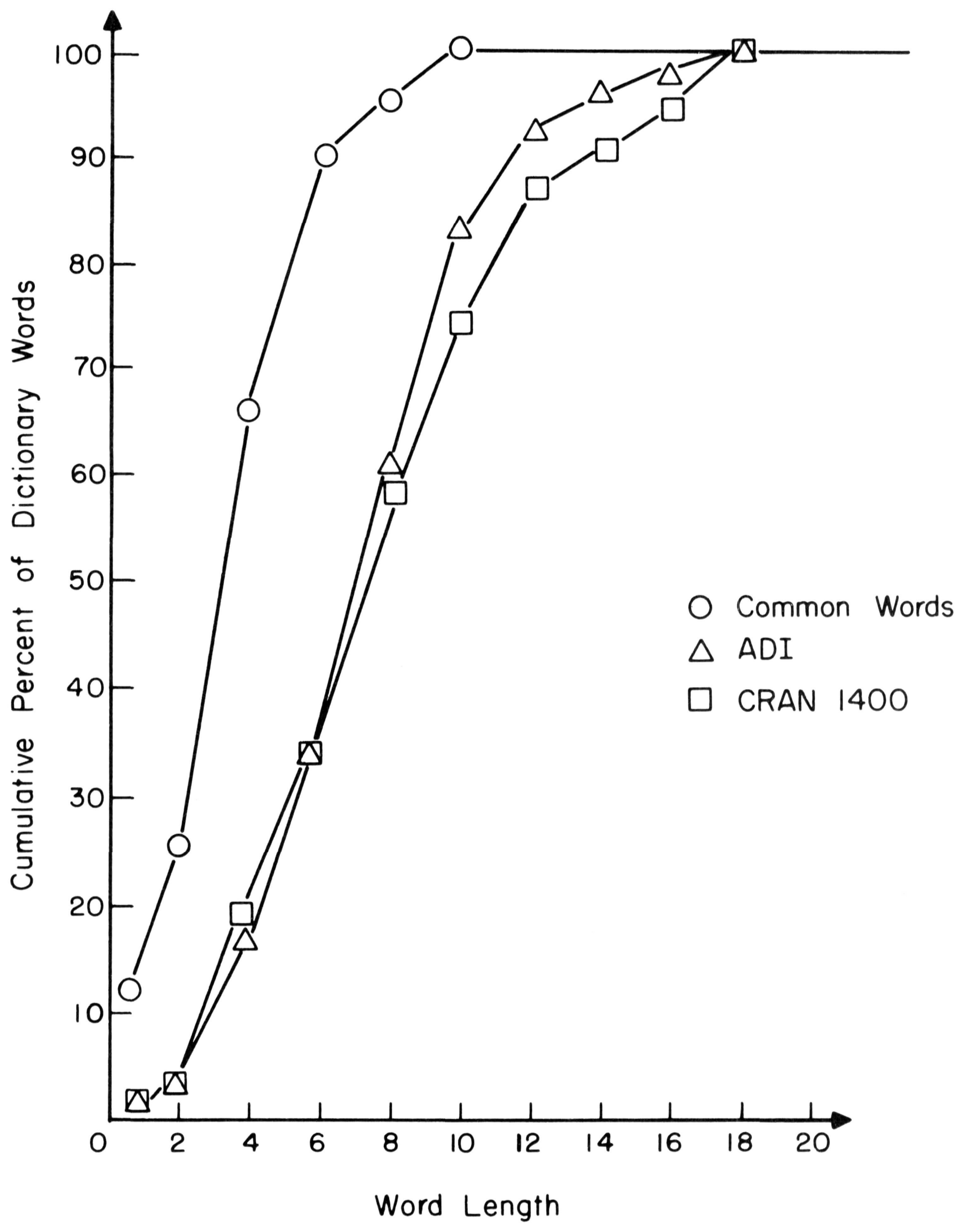
1) is not available on a keypunch or teletype,

2) will not propagate zeroes,



Distribution of Dictionary Words
According to Their Lengths

Fig. 2



Cumulative Distribution of Dictionary Words
According to Their Lengths

Fig. 3

3) will generate a few carries during key formation, and

4) has the majority of its bits equal to 0, so their positions may be filled.

A likely candidate for the S/360 is 01000101.

Three basic methods of generating virtual hash addresses — addition, multiplication, and division — are studied. The first and second provide contrasting ways of forming the single fullword keys. The second and third differ in the way the hash address is computed from the key. Variations of each basic method are also tested to try to improve speed, programming ease, or collision-storage properties.

1. Addition Methods

   AC — addition and center
   The fullwords of characters are logically added to form the key. The key is squared and the centermost bits are selected as the major. The minor is obtained from bits on both sides of the major.

   AS — addition with shifting
   Same as AC, except the second, third, etc. full words are shifted two positions to the left before their addition in forming the key. (An attempt to improve collision-storage properties)

   AM — addition with masking
   Same as AC, except the second, third, etc. fullwords have certain nonsignificant bits altered by masks before their addition in forming the key. (An attempt to improve collision-storage properties)

2. Multiplication Methods

   MC — multiply and center
   The fullwords of characters are multiplied together to form

the key. The center bits of the previous product are saved as the multiplier for the next product. The key is squared and the centermost bits selected as the major. The minor is obtained from the bits on both sides of the major.

MSL — multiply and save left
Same as MC, but during formation of the key, the high order bits of the products, rather than the center, are used as successive multipliers. (An attempt to improve speed)

MLM — multiply with left major
Same as MC, but taking the major from the left half of the square of the key and the minor from the right half. (An attempt to improve speed)

3. Division Methods

DP — divide by prime
The fullwords of characters are multiplied together to form the key. The center bits of the previous product are saved as the multiplier for the next product. The key is divided by the length of the virtual hash table — a prime number in this case — and the remainder used as the virtual hash address. The major is drawn from the left end of the virtual address and the minor from the right.

DO — divide by odd number
Same as DP, except using a hash table whose length is odd. (An attempt to provide more flexibility of hash table sizes)

DT — divide twice
Same as DP, except two divisions are made. The major is produced by dividing the key by the actual hash table size. The minor results from a second division. Primes are used throughout as divisors. (An attempt to improve storage-collision properties)

C) Evaluation

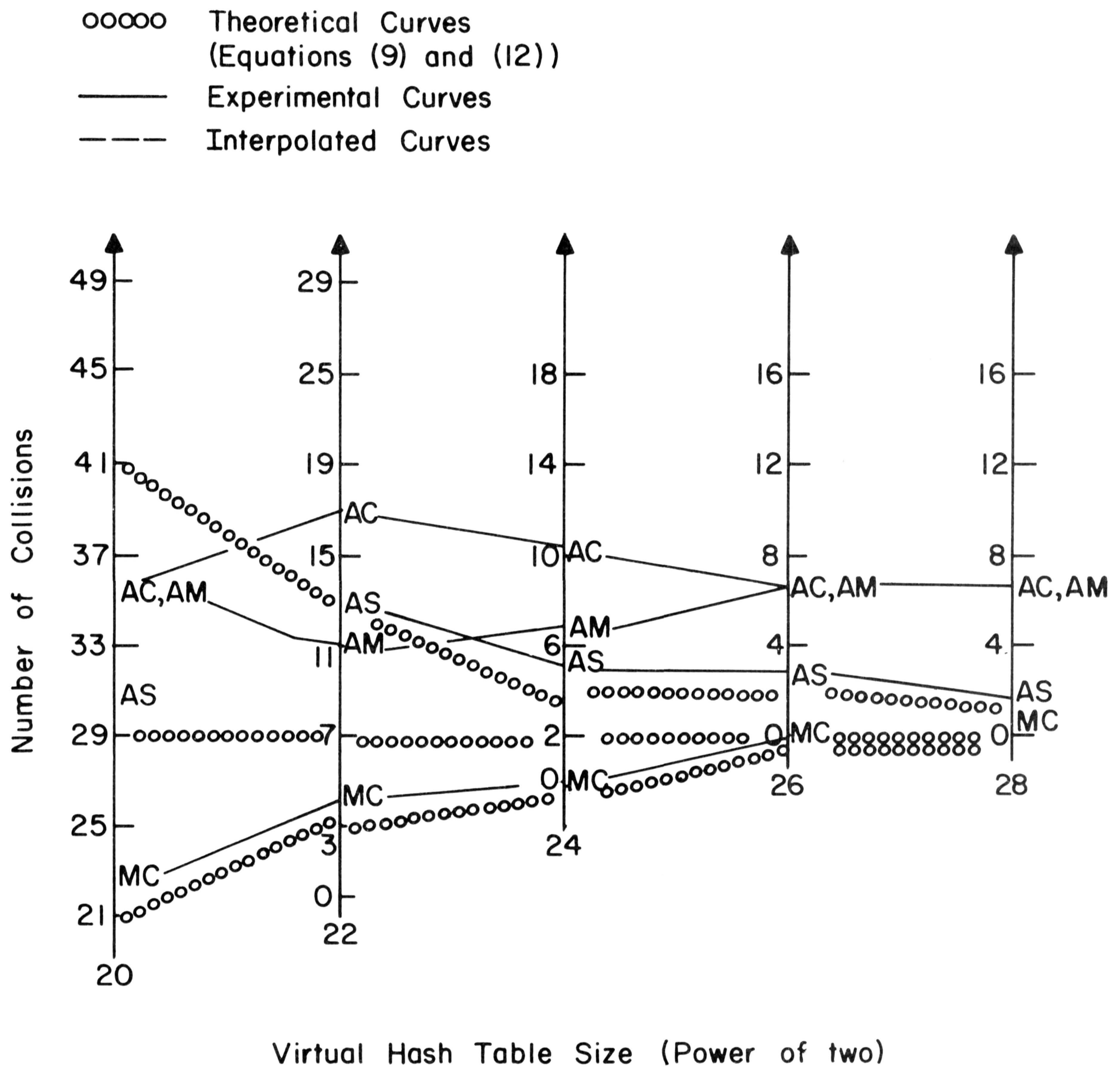
In the experiments to evaluate each variation of the above hash schemes, the size of the virtual hash table varies from 2^{20} to 2^{28} slots. The actual hash table varies in size from 2^{12} to 2^{14} slots. Bump table space is used as needed. The tables are filled by the words from either the ADI or CRAN dictionaries and the collision and storage statistics taken. Because good collision properties are most important, they are examined first. The storage properties are dealt with later.

The number of collisions obtained from each scheme versus the virtual table length is plotted in Figs. 4 to 7. The ADI dictionary is shown in Figs. 4 and 6, and the CRAN in Figs. 5 and 7. The circled lines correspond to curves generated from equations (9) and (12). The horizontal one shows the expected number of collisions and the lines above and below it enclose a 95% confidence interval about the expected curve. In other words, if an algorithm is generating random addresses, the probability is 95% that the curve for that scheme lies between the heavy lines.

Consider Figs. 4 and 5 showing the results for all the addition methods and the MC variation of the multiplication variation. The AC and MC algorithms differ only in that addition is used in forming the key in the first one and multiplication in the second one. Yet the curves are spectacularly different. The result seems to have the following explanation.

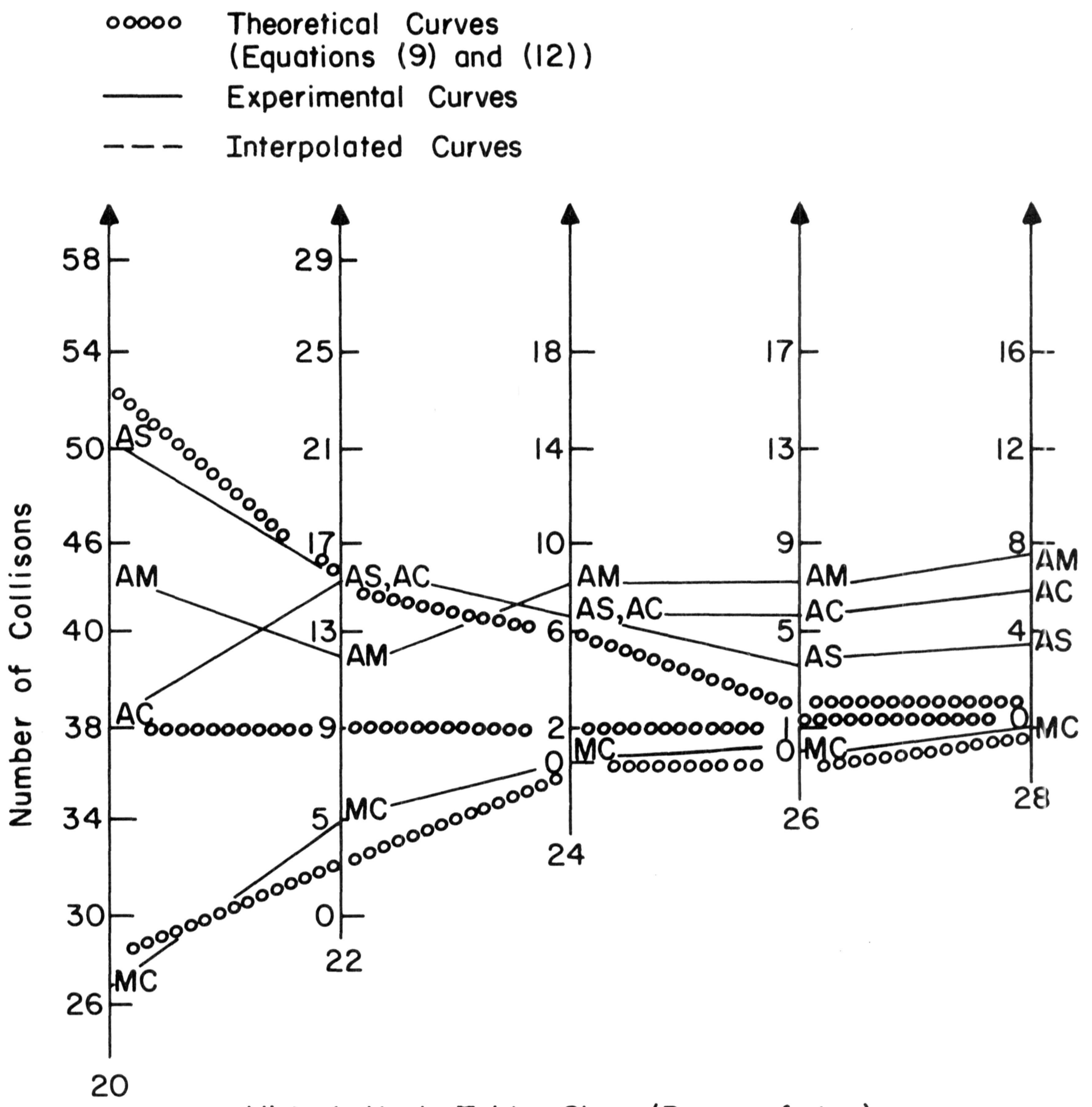
The purpose of a hash address computation is to generate a random number from a string of characters. If the bits in the characters are as varied as possible, then the algorithm has a headstart in the right direction. However, the S/360 bit patterns for the alphabet and numbers are:

A to I	1100 xxxx
J to R	1101 xxxx
S to Z	1110 xxxx
0 to 9	1111 xxxx



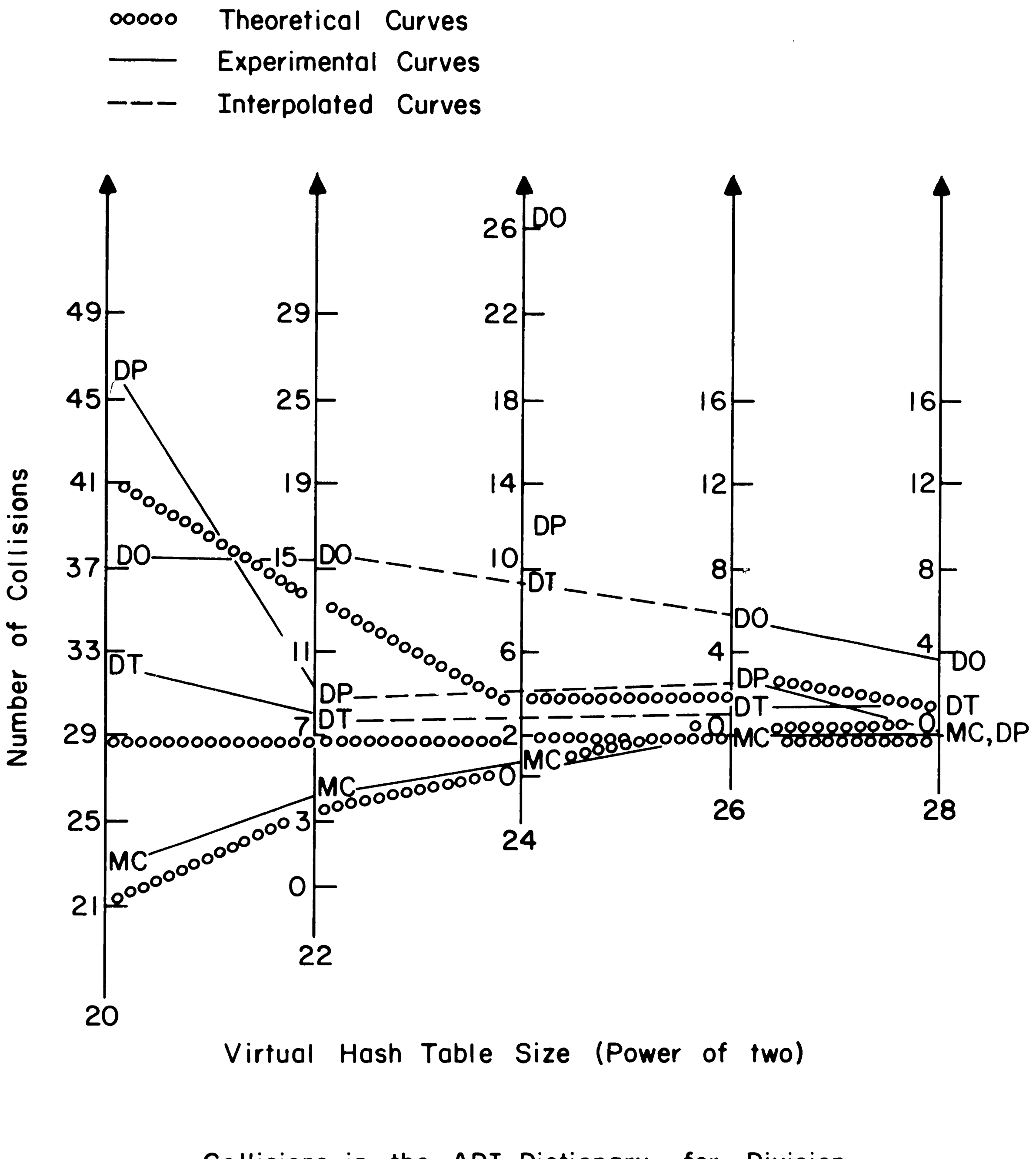
Collisions in the ADI Dictionary for Addition
and Multiplication Hash Schemes

Fig. 4



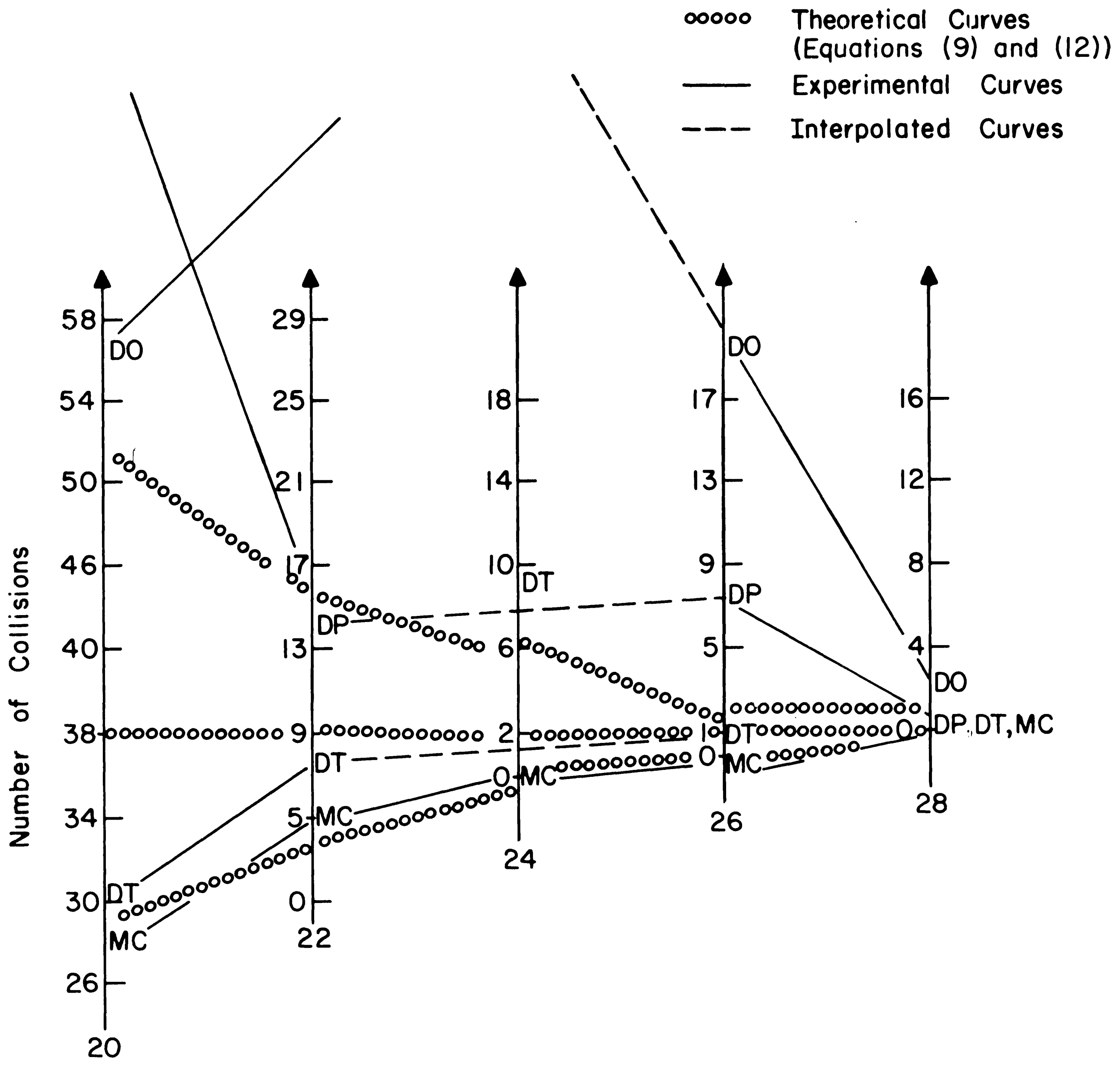
Collisions in the CRAN Dictionary for Addition and Multiplication Hash Schemes

Fig. 5



Collisions in the ADI Dictionary for Division and Multiplication Hash Schemes

Fig. 6



Collisions in the CRAN Dictionary for Division and Multiplication Hash Schemes

Fig. 7

In each case the two initial bits of a character are 1's so that in any given word $\frac{1}{4}$ of the bits are the same.

In forming a key, the successive additions in the AC algorithm may obscure these nonrandom bits if sufficient number number of carries are generated. However, the number of additions performed is usually small — 2 or 3 — and it appears that the patterns are not broken sufficiently. The MC algorithm uses multiplication to form its keys which involves some 31 additions — certainly enough to make the resulting key random.

The multiplications in the MC algorithm are costly in terms of computation time. Therefore the AS and AM algorithms are tried. These addition variants try to hasten the breakup of the nonrandom bits by shifting and masking respectively. Although these variants reduce the number of collisions somewhat, none of the addition schemes could be called random. Typically a few words are singled out at some point and continue to collide regardless of the length of the virtual address. Several collision pairs are listed below. Note the similarities between the words.

COUNT	—	SOUND
WORTH	—	FORTY
TOLERATED	—	TELEMETER
WHEEL	—	SHEET

Consider the multiplication algorithms. During key formation, the process of saving the center of successive products adds to the computation time. The MSL variation attempts to remedy this by saving only the high order bits between multiplications (on the S/360 this means saving the upper 32 bits of the 64 bit product). This method is so inferior that its collision graph could not be included with the others. The poor results stem from the fact that characters at the end of fullwords have little effect on the key and that the later multiplications swamped the effects of the earlier ones. Examples of collision pairs are given below. For convenience the fullwords are separated by blanks.

CERT AINT Y	—	CERT AINL Y
PREV ENTE D	—	PRES ENTE D
HEAV ING	—	HEAT ING
EXPE NSE	—	EXPA NSE
CHAR TER	—	CHAP TER

The MC and MLM variants are identical with respect to collision properties. In general these algorithms produce good results, reducing the number of collisions to zero in both dictionaries. The collision curve is always beneath the expected one.

Consider Fig. 7 and 8 showing the results for all division methods and the MC method. All of the division algorithms display a distinct rise in the number of collisions when the virtual table size is near 2^{24} — regardless of the dictionary. The majority of the colliding word pairs are 4 character words having the same two middle letters. This brings to light a curious fact about division algorithms. For virtual tables, the divisor of the key is large and the initial few bits determine the quotient, leaving the rest for the remainder. For words of less than 4 characters (which require no multiplications during key formation), dividing by 2^{24} is equivalent to selecting the last 3 characters of the word as the hash address. Because the divisors are not exactly equal to 2^{24}, only the two middle characters tend to be the same. Examples are:

DEAL — BEAR

TOOK	—	SOON
HELD	—	CELL
VERB	—	TERM

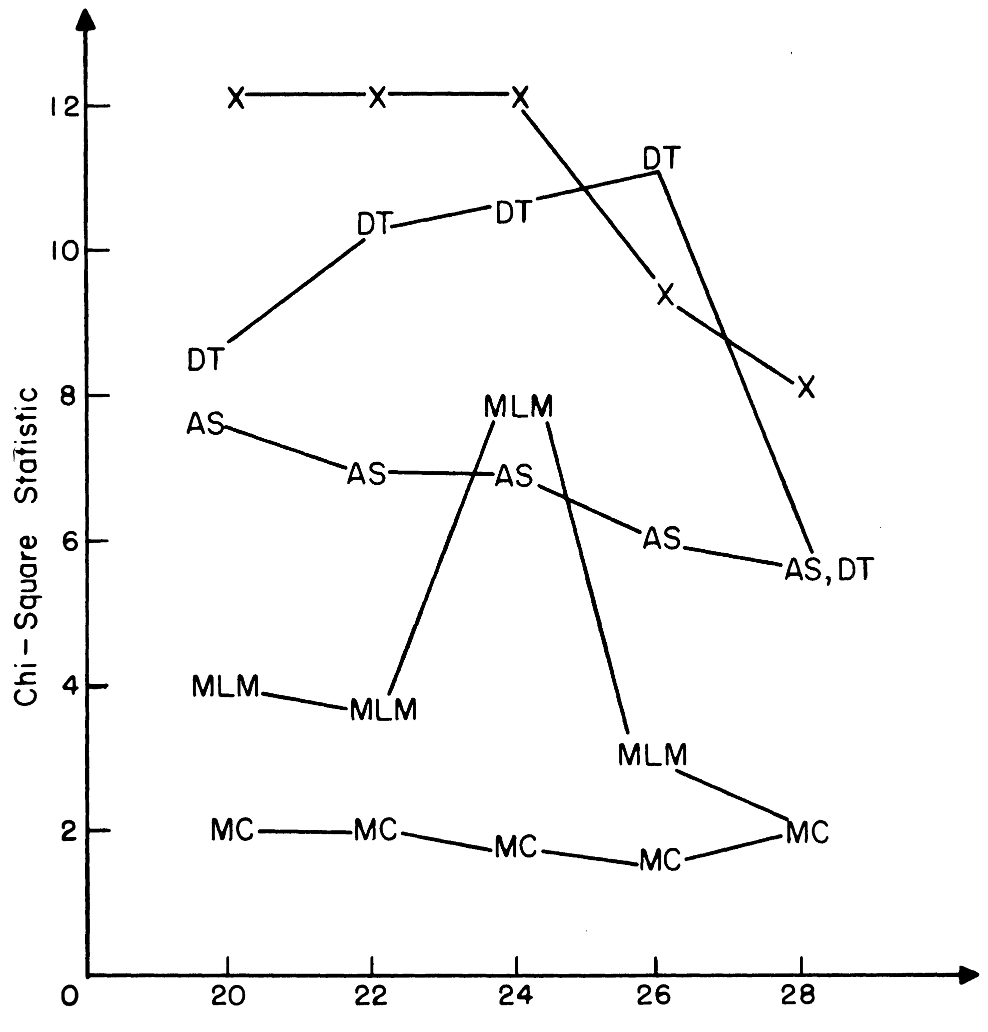
This phenomenon apparently continues for table sizes around 2^{26} and 2^{28}, but there are few or no words of 4 characters or less which agree in 26 or 28 bits. For divisors smaller than 2^{24}, a larger part of the key determines the quotient and apparently breaks up the pattern.

Because the above effect occurs only for $V = 2^{24}$, these points are passed over on the graphs.

In general, the DT algorithm is superior to the rest of the division methods, mostly because each of its two divisors is smaller than those used in other methods. Prime numbers seem to produce better results than other divisors.

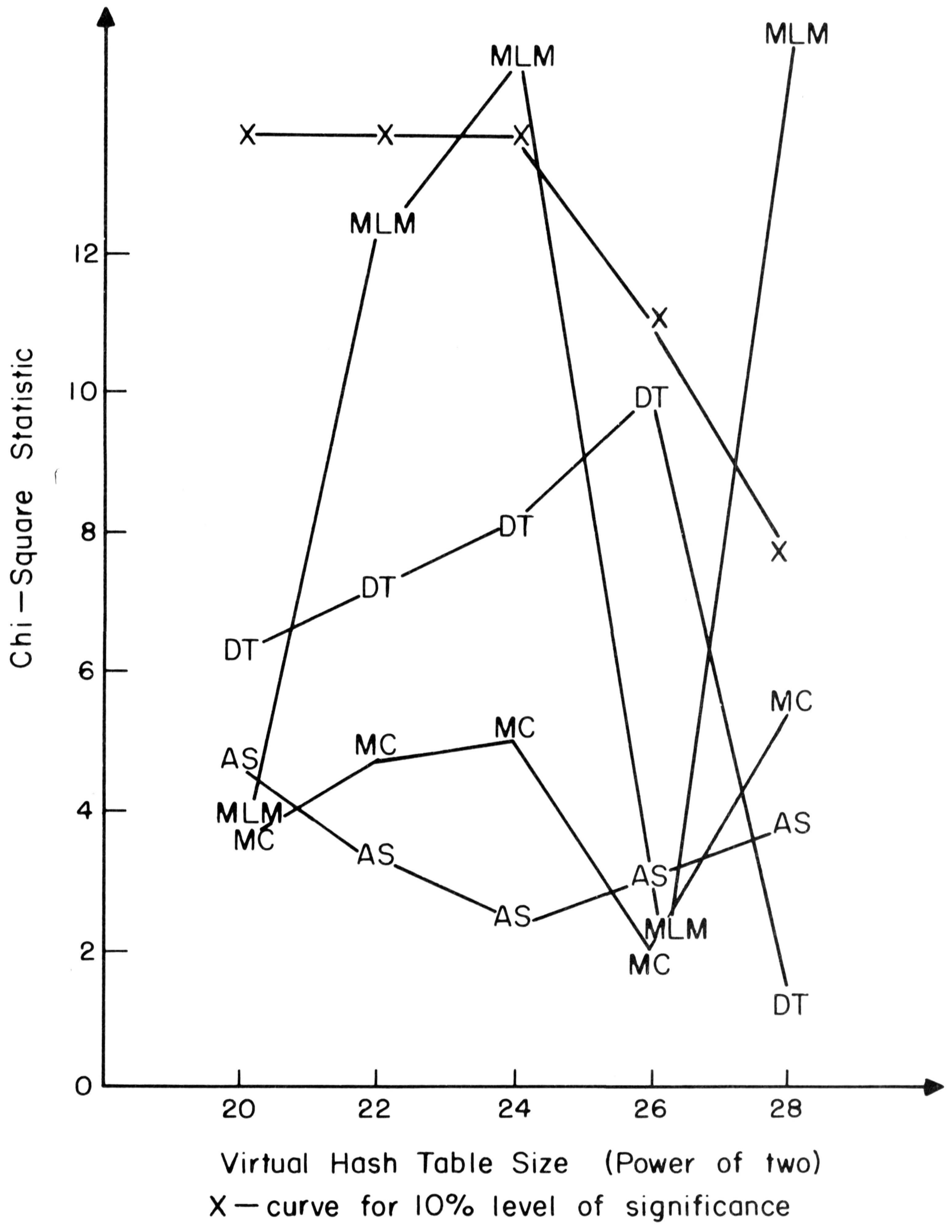
On the basis of collision properties, the MC, MLM, DT, and possibly AS algorithms are the best. Storage-search evaluations are included for these methods only.

The experiments with each hash coding method also include counting the frequency of various length of collision chains. Here a collision chain refers to chains of words producing the same major. The frequency counts are compared with the expected counts given by equation (5). The comparison is in terms of a chi-square goodness of fit test with a 10% level of significance. Figs. 8 and 9 show the results of this test for each dictionary. Included in the graphs is the line corresponding to the 10% level of significance. If the major portions of the hash addresses are really random, there is a probability of 0.90 that the 10% line will lie above the curve for the algorithm tested.



Deviations of Storage — Search Properties from Expected Values for Storage Hash Schemes using the ADI Dictionary

Fig. 8



Deviations of Storage—Search Properties from Expected Values for Selected Hash Schemes using the CRAN Dictionary

Fig. 9

Consider the MC and MLM algorithms which differ only in that the major is selected from the center and left of the virtual address. From the graphs, it is clear that the multiplication methods produce their most random bits in the center of their product. This is somewhat as expected because the center bits are involved in more additions than other bits.

The division algorithm, which had fairly good collision properties, seems to have rather mediocre storage properties. This is probably due to the same causes as the collision problems, but working at a lower level, and not affecting the results as much.

The AS curve is included simply for completeness. The scheme displays a well behaved storage curve, but unfortunately it has poor collision properties.

In summary, the MC scheme seems to be the best for both dictionaries in terms of collision and search properties. In terms of computing time, the method is more time consuming than the addition methods, but less expensive than the division methods. The differences in computation times is not an extremely big factor. All methods required from 35 to 55 microseconds for an 8 character word on the S/360/65. The routines are coded in assembly language and called from a Fortran executive. The times above include the necessary bookkeeping for linkage between the routines.

5. A Practical Lookup Scheme

A) General Description

The lookup scheme described below is designed for use with dictionaries of about 2^{15} words. The virtual table size selected is 2^{29} and the actual table size is 2^{15}. On the basis of the results presented in previous sections, when the dictionary is full, it is expected that

1) 36.8% of the hash table will be empty,

2) 36.8% of the hash table will be single entries,

3) the bump table will require $(0.632)2^{15}$ entries,

4) 1 collision is expected,

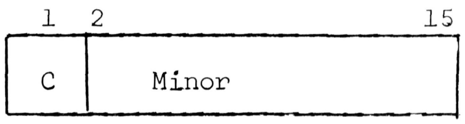
5) the probability of 5 or fewer collisions is 0.999, and

6) the average lookup will require 2.13 probes.

B) Table Layout

In all previous discussions a dictionary entry has included a minor and a concept number. A concept number is simply a unique number assigned to each word. The hash address of a word is also unique, and hence can be used. There is no need to store and use a previously assigned concept number.

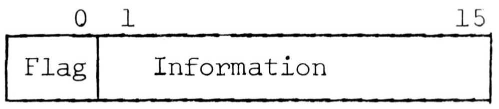
A dictionary entry contains a 14 bit minor and a single bit indicating whether the word is common or noncommon:



C = 0 implies the word is common

C = 1 implies the word is noncommon.

A hash table entry contains 16 bits arranged as:



Flag = 0 implies that the information is a dictionary entry

Flag = 1 implies that the information is a pointer to the bump table

Words that have the same major are stored in a block of consecutive locations in the bump table. This eliminates the need for pointers in the collision "chains". A bump table entry also has 16 bits structured as:

0	1	2	15
End	C	Minor	

End = 0 implies that the entry is not the last in the collision block

End = 1 implies that the entry is the last in the block.

Some convention must be adopted to signify an empty hash table slot. A zero is most convenient in the above scheme. Unfortunately a zero is also a legitimate minor. However, to cause trouble the word generating the zero minor would have to be a common word and a single table entry (zero minors in the bump table are no problem). Hopefully this occurs rarely because of the size of the minor (14 bits) and the small number of common words. However, even if this combination of circumstances occurs, the common word could be placed in the bump table anyway.

In designing the tables, it is important to make the hash table entries large enough to accommodate the <u>largest</u> pointer anticipated for the bump table. For the above scheme, the expected bump table size is less than 2^{15} so that the 15 bits allocated for pointers is sufficient.

C) Search Considerations

The number of probes needed to locate any given word depends on the place that the word occupies in a collision block. The average search time is improved if the most common words occupy the initial slots in each block. A study of ADI text yields the following statistics.

Division of Words by Category	
Number of Words	Percent of Total
17270 Total words	100.0
8716 Common words	50.5
8554 Noncommon words	49.5

Distribution of Lengths						
Number of Characters	All Words	Percent	Common Words	Percent	Non-common Words	Percent
1-4	10145	58.8	8057	92.5	2097	24.5
5-8	4630	26.8	627	7.2	4003	46.8
9-12	2249	13.0	32	0.3	2217	25.9
13-16	221	1.3	0	0.0	221	2.6
17-20	11	0.1	0	0.0	11	0.1
21-24	5	0.0	0	0.0	5	0.1
Totals	17270	100.0	8716	100.0	8554	100.0
Average Length	6.3		4.3		8.3	

Using the categorical information, it appears that in filling the hash-bump tables, the common words should be entered first. Within each category, all words should be entered in frequency order if such information is known. If frequency information is not available, the distribution by lengths can be used as an approximation to it. For common words, this means entering the shorter words first. For noncommon words, the words of 5 to 8 characters should be entered first.

The greater the number of single entries, the greater the average search speed. Fig. 10 shows the fraction of single entries (F_1) and fraction of empty slots (F_o) for various load factors. The fraction of single entries $F_1 = \alpha e^{-\alpha}$ reaches a maximum for $\alpha = 1$, but since the slope of the curve is small around this point, and load factor in the interval (0.8, 1.2) is practically the same. Table usage is better, however, for the larger values of α. These facts imply that scatter storage schemes make most efficient use of space and time for $\alpha = 1$.

Most text words can be assumed to be in the dictionary. Thus the order of comparisons during lookup should be:

Hash Table Scan

1) check minor assuming the text word is a common word

2) check minor assuming the word is noncommon

3) check if the entry is a pointer to the bump table

4) check if the entry is empty

First Bump Table Entry (must be at least two)

5) check minor assuming the word is a common word

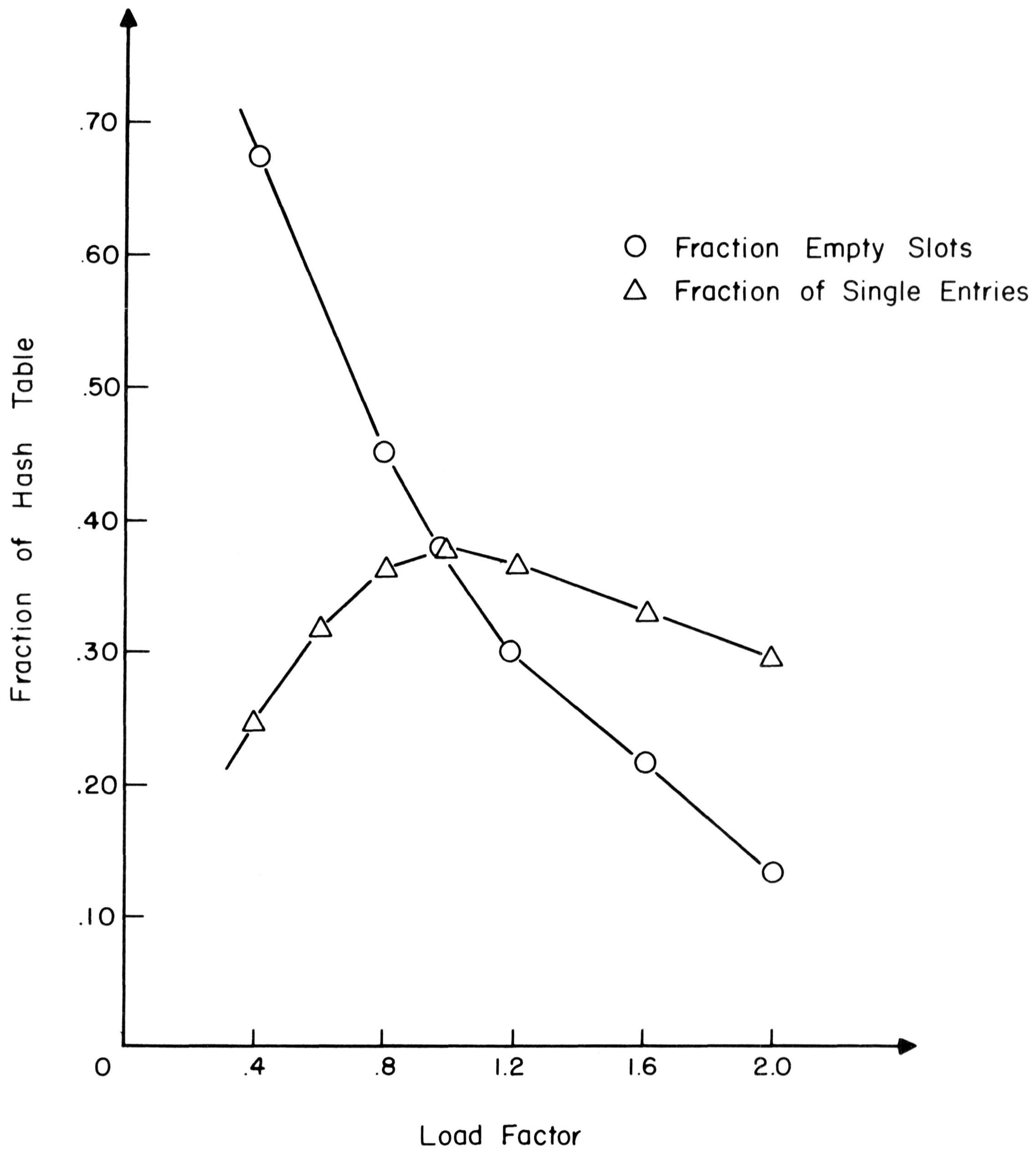
6) check minor assuming the word is noncommon

Other Bump Table Entries

7) check minor assuming the word is noncommon

8) check minor assuming the word is common

9) check if at end of collision block.



Theoretical Hash Table Usage

Fig. 10

The search pattern can be varied to take advantage of the storage conditions. For example, if all common words are either single entries or the first element of a collision block, then step 8) may be eliminated.

D) Performance

The lookup system described above has been implemented and tested on the IBM S/360/65. A modified form of the MC algorithm is used to compute a 29 bit virtual address and divide it into a 15 bit major and a 14 bit minor. The modification is the inclusion of a single left shift of the fullwords of characters during key formation. This breaks up certain types of symmetries between words such as WINGTAIL and TAILWING. Without this, such words will always collide. The hash-bump tables were filled with entries from the ADI dictionary — common words first, followed by noncommon words. The shortest words were entered first. A comparison of the expected and actual results follows.

α = .239	Expected	Actual
Number of empty table slots	25810	25762
Number of single entries	6161	6250
Number of collision blocks	797	756
Longest collision block	4	4
Average length of collision blocks	2.1	2.1
Size of bump table	1663	1572
Number of collisions	.06	1.33
Average probes per lookup	1.33	

To obtain the actual lookup times 627 words were processed. The words were read from cards and all punctuation removed. Each word was passed to the lookup program as a continuous string of characters with the proper number of fill characters added. The resulting times are given below (in microseconds):

Category of Words	Number of Words	Percent of Total	Average Time	Standard Deviation	Average Probes
All	627	100.0	57.9	11.7	1.18
Common	288	45.9	49.9	6.7	1.12
Noncommon	338	53.9	64.7	10.7	1.24
Not found*	1	0.2	53.1	0.0	1.00

* A larger sample with less accurate timings indicates that the average time for words in this category is about 62 microseconds (standard deviation 26).

The time to compute a hash address depends on the length of the word. Let n be the number of S/360 fullwords needed to hold these characters. The time to form the initial address is

$$I(n) = 34.5 + 10.2\,(n - 1) \qquad \text{microseconds.}$$

The average total lookup time, then, is

$$T = I(n) + cP$$

where c is the average time per probe into the table space and P is the average number of probes. For the words in the experiment $n = 2.32$ (average), $I(n) = 40.3$, and $T = 57.9$ so that each probe required about 15 microseconds.

E) Comparisons

Timing information for other lookup schemes is difficult to obtain. A tree structured dictionary is used for a similar purpose at Harvard. Published information indicates 6pq microseconds are needed to process p words in a dictionary of q entries. This time is for the IBM 7094. Translating this time to the S/360/65, which is roughly 4 times faster, and using the ADI dictionary (q = 7822), it appears that each lookup averages 11,000 microseconds. Exactly how much computation and input-output this includes is unknown.

6. Extensions

A) Larger Dictionaries

As more words are added to the dictionary, the size of the virtual address must increase in order to prevent collisions. As a result, the number of bits per table slot must also increase in order to accommodate the larger minors and pointers that are used. For a fixed sized hash table, the number of entries in the bump table grows as new words are added. At some point the space required for tables will exceed the amount of core allotted for dictionary use. To salvage the scheme, it may be possible to split the bump table into parts — one part for more frequently used words and one for words in rather rare usage. During dictionary construction common words are entered first, then noncommon, then rare. When a rare word must be placed in a collision block, a marker is stored instead, and the item is placed in the secondary bump table. Presumably the nature of the words in the second bump table will make its usage rather infrequent, thus saving accesses to auxiliary storage to fetch it.

B) Suffix Removal

Many dictionary schemes store only word stems; the lookup attempts to match only the stem, disregarding suffixes in the process. This is not easily done with scatter storage schemes. One solution is to try to remove the suffix after an initial search has failed. Each of the various possible stems must be looked up independently until a match is found. Another solution is to use a table of correspondences between the various forms of a word and its stem. The concept number could be used as an index on this table containing pointers to information about the actual stem. A thesaurus lookup can be handled the same way.

7. Conclusions

Virtual scatter storage schemes are well suited for dictionaries, having both rapid lookup and economy of storage. The rapid lookup is due to the fact that the initial table probe limits the search to only a few items. The space savings come from the fact that the actual character strings for words are not part of the dictionary. The schemes depend heavily on a good algorithm for producing random hash addresses. The theory developed in Sections 2 and 3 gives a basis for judging the worth of proposed algorithms.

For any particular application, the table organization may vary to suit different needs and to store different information. However, the advantages of scatter storage schemes are still present.

References

[1] Salton, G., A document retrieval system for man-machine interaction, Association for Computing Machinery, Proceedings of the 19th National Conference, Philadelphia, Pennsylvania, August 25-27, 1964, pp. L2.3-1 — L2.3-20.

[2] McIlroy, M. D., Dynamic Storage Allocation, unpublished manuscript, Bell Telephone Laboratories, Inc., 1965.

[3] Morris, R., "Scatter Storage Techniques", Communications of the ACM, January, 1968.

[4] Maurer, W. D., "An Improved Hash Code for Scatter Storage", Communications of the ACM, January, 1968.

[5] Johnson, L. R., "Indirect Chaining Method for Addressing on Secondary Keys", Communications of the ACM, May 1961.