Selecting a Hashing Algorithm

B. J. MCKENZIE, R. HARRIES AND T. BELL

Department of Computer Science, University of Canterbury,

Christchurch, New Zealand

SUMMARY

Hashing is so commonly used in computing that one might expect hash functions to be well understood, and that choosing a suitable function should not be difficult. The results of investigations into the performance of some widely used hashing algorithms are presented and it is shown that some of these algorithms are far from optimal. Recommendations are made for choosing a hashing algorithm and measuring its performance.

KEY WORDS hashing symbol tables

INTRODUCTION

Hashing has always had many applications in computing. Moreover a great deal has been written about the topic. We might, then, reasonably assume that applications of hashing in widely distributed software will use algorithms that have been carefully selected and equally carefully tested. The fact that we can find examples where this appears not to be the case is perhaps more a reflection of the fact that some aspects of the behaviour of hashing algorithms are somewhat counter-intuitive, than of negligence on the part of the programmers. The discoveries reported in this paper arose because a student was adding to the Amsterdam Compiler Kit [1], a library of compiler tools, and proposed to use the hashing routine already in the library. A check of the distribution of hash values produced by the routine provoked a series of further discoveries.

A SUSPECT HASHING ALGORITHM

In the description of the algorithms we shall use the following:

(i) The identifier (or word) being hashed consists of n characters whose ASCII values are c_1 , .. c_n. These array elements are stored as 'signed bytes'. That is to say, the most significant bit is taken as a sign bit. The other variables are stored as (32 bit) integers. When a 'signed byte' is used in an arithmetic expression or is stored in an integer, it is extended to 32 bits in length by propagating the sign bit. Of course with the standard 8-bit representation of ASCII this distinction is irrelevant since the most significant bit is always zero, nonetheless it will be seen later that there are circumstances in which the fact that this form of representation has been used is relevant.

(ii) The function BITS(v, n) returns an integer whose least significant n bits are the same as the least significant n bits of v, and whose other bits are zero.

(iii) The functions AND(x, y), OR(x, y), and XOR(x, y) perform bitwise logical 'and', 'inclusive or' and 'exclusive or' on their arguments and return the result.

(iv) A hashing algorithm is an implementation of a function whose range is a set of integers. In what follows the range is taken to be 0 to $N–1$ (for some natural number N). Such an algorithm will be said to hash to N buckets.

(v) When distinct inputs hash to the same value a 'collision' is said to occur. When the hash value is used as a key (for example, as an index into a table) some method needs to be adopted for dealing with collisions. In what follows it is assumed that the technique known as 'separate chaining' [2] is employed, and that there is effectively infinite memory available for storing the chains.

The algorithm which led to the present investigation was:

```
Algorithm ACK:
  m1 = 171
  mi = BITS(77.mi-1 + 153, 8)   for 2 ≤ i ≤ 16
  h0 = 0
  hi = hi-1 + XOR(ci, mi)  for 1 ≤ i ≤ 16
  hi = hi-1  for i > 16
  H = BITS(hn, 8)
```

The mask array elements m_i in this algorithm are stored as 'signed bytes'. The process used to generate these pseudo-random numbers is a linear congruential generator. The routine ACK was used to hash 36 376 identifiers collected from a large number of programs written in C. As a check that there was not some curious property of C identifiers which did not hold generally, it was also used to hash 24473 words from a Unix dictionary. Figure 1a gives the resulting plots of the number of identifiers (and words) hashed to each bucket. It had been expected that this distribution would be more or less flat, and its marked U-shape was surprising.

EXPLANATIONS

Let us consider first a hashing scheme in which we merely added the ASCII values of the characters. Suppose all the identifiers were sorted by length and that we were to hash first all the identifiers of length 1, then all the identifiers of length 2, and so on, plotting the distribution of hash values each time. For identifiers of length 1 we shall produce a distribution in which for some values of x in the range $0 \le x \le 127$, y takes the value 1, and everywhere else y is zero. For identifiers of increasing length we should get distributions which are increasingly more humped (i.e. have a lower coefficient of variation). The peak of the hump

for length n will have an x coordinate equal to roughly n times the mean for length 1. The height of the hump depends in part inversely on the coefficient of variation for that length, and in part on the relative frequency of identifiers of that length. The distribution obtained when *all* of the identifiers are hashed is the sum of these separate distributions. Some of these distributions are shown in Figure 2.

A considerable part of the explanation of the behaviour of algorithm ACK depends on the observation that the upper case letters all have ASCII representations in which the most significant three bits are 010, while the lower case letters have correspondingly 011. Whenever we apply 'XOR' to a letter and a mask byte (both regarded as signed bytes), therefore, the resulting integer will have the same sign as the mask byte, since the most significant bit of the letter is zero. Also, because most characters in the identifiers (or words) are lower case letters, the next two bits are usually ones, and so only the five least significant bits contribute to the spread. If the mask array contains any negative values, some of the distributions will have their means (and peaks) moved to the left, some even to the left of the y-axis. Indeed, if the masks are truly random, and the mask array is sufficiently long, we should expect the peaks to get closer and closer to $x = 0$ as the lengths get greater. The distribution for *all* of the identifiers is therefore markedly peaked in the vicinity of $x = 0$. But the final step in the algorithm is to take the low order 8 bits of the sum. If the result were regarded as a signed byte the values would lie in the range -127 .. +127 and would be markedly peaked about $x = 0$. Since the result is instead regarded (effectively) as an unsigned byte, the values lie in the range 0 .. 255 with a U-shaped distribution.

SOME OTHER HASHING ALGORITHMS

The results described above made it seem worthwhile to examine the hashing algorithms employed in some other widely used software. Table A contains a description of the algorithms examined.

The algorithm ETH produced an anomaly, namely that the number of identifiers with hash value 1 greatly exceeded the number for any other hash value The bulk of the identifiers contain three or more characters. On average, in one case in 257 the value of h_i prior to dealing with the last character will be such that h_i *MOD 257* will be zero. Consequently for one word in 257 the hash value will be 1, and this is independent of the number of buckets. Modification of the algorithm to read

```
ETH-new:
  h0 = 1;    hi = ci.((hi-1 MOD 257) + 1)    for 1 ≤ i ≤ n
  H = hn MOD 1699
```

improved the distribution. (See Figure 1b.)

The distribution of the items into buckets by the other algorithms is shown in Figure 1.

QUANTIFYING THE ALGORITHMS

Selection of a hashing algorithm is presumably done on the basis that that algorithm is 'better' than others. The question is: what criteria are to be used in deciding what is 'better'? Two criteria seem plausible: (i) the degree to which the algorithm uniformly distributes candidate keys over the possible values; (ii) the speed with which the algorithm executes. For both of these parameters it is possible either to develop a theoretical measure or to make empirical measurements, and for both criteria we took the latter course.

The distribution could be subjected to a chi-squared test taking as null hypothesis H_0: *the algorithm produces a uniform distribution (and the observed deviations from a uniform distribution occur by chance).* The chi-squared test gives us a probability that H_0 is true. Unfortunately the probabilities determined this way are so small that they are of doubtful accuracy and are not in any case very easily assimilated. However if, as is the case here, the number of degrees of freedom, ν, is reasonably large ($\nu > 100$), then $Q(\chi^2|\nu) \approx Q(x_1)$, where $x_1 = \sqrt{(2.\chi^2)} - \sqrt{(2.\nu - 1)}$. The function Q measures the tail of the cumulative distribution function; that is $Q(X) = 1 - P(X)$ where P(X) is the cumulative distribution function for X (see [3].) Consequently we can take x_1 as the measure of the 'goodness' of the distribution.

Another approach to measuring the 'goodness' of the distribution relies on the observation that the cost of using a hash table depends in part on the number of probes required to locate identifiers in it. The number of probes required to independently retrieve all the identifiers in a chain of length n is $n(n+1)/2$. If we sum this expression over all chains we find the number of probes required to locate all the identifiers in the table. Suppose there are W identifiers altogether and that the hashing function has a range from 0 to $N-1$. Let

$$S_N = \Sigma f_{i,N}(f_{i,N} + 1)/2$$

$$e_N = W/N$$

$$T_N = N\, e_N(e_N + 1)/2 = W(W+N)/2N$$

$$R_N = S_N / T_N$$

where the sum runs from 0 to $N-1$, and $f_{i,N}$ is the number of identifiers hashing to i when the hashing function range is 0 to $N-1$.

Then R_N is the ratio of the number of probes actually required to retrieve all of the identifiers to the number required if the distribution were uniform. Values of R_N were determined for each algorithm and for various values of N by measuring values of $f_{i,N}$.

Execution speeds were also measured. Originally the algorithm ETH was written in Modula-2. For this investigation it was rewritten in C, since all the others were already in that language. In these latter cases various coding techniques had been used to decrease execution time. For example, the BITS function was implemented in line by means of a logical *and* with a mask; multiplication by powers of two was done by shifting; register variables were used where appropriate. These techniques were applied consistently throughout.
The results of these investigations are contained in Table B.

It is reassuring to notice that rankings by x_1 and by R_N produce similar results. These parameters are essentially measuring the degree to which the algorithm distributes items evenly over the buckets. On the other hand t, which measures the intrinsic time to hash an item, and is independent of the spread over buckets, gives a different ranking. Clearly a compromise must be made between the time taken to calculate the hash function, and the time spent resolving collisions produced by an inferior function.

COMPARISON OF THE ALGORITHMS

An ideal hashing function would produce a value of 1 for the ratio R_N. It is easy to show that in the worst possible case, where every identifier hashes to the same value, $R_N \approx N$. In general we find that as N increases so does R_N. Consequently for a fair comparison of the algorithms one should use the same value of N for each. Consideration must, however, be given to whether the values of N embodied in the algorithms as given can be changed without significantly altering the performance of the algorithm in some other way. In some cases the value was chosen to be a power of two so that the MOD function could be calculated

rapidly. In other cases the number of buckets was presumably chosen because it was felt that that value would give a 'better' distribution.

Reflection on these issues leads to curiosity as to the degree to which the performance of the algorithms is susceptible to changes of N. It was decided to plot values of R_N against N for each algorithm. From this point of view the algorithms GNU-cpp, GNU-cc1, PCC, CPP, C++ and Icon all have the form

$$h_i = k.h_{i-1} + c_i \quad \text{for } 1 \le i \le n$$
$$H = h_n \text{ MOD } N$$

where k is 1 for Icon, 2 for PCC, CPP and C++, 4 for GNU-cpp and 613 for GNU-cc1. Certain other minor changes were also made to remove inessential differences:

(i) The function BITS was not used.

(ii) Calculation of H was in all cases done thus:

$$H = \text{if } (h_n \text{ MOD } N < 0) \text{ then } N + h_n \text{ MOD } N \text{ otherwise } h_n \text{ MOD } N$$

Figure 3 shows plots of R_N against N for $k = 2,\ 4,\ 13$ and 613.

For purposes of comparison we also give plots for ACK and ETH-new. Figure 4 is a combined graph in which in some cases only a selection of the points have been plotted. In the case of GNU-cpp and PCC only odd values of N are given; for ETH-new only prime values. The vertical line at $N = 1787$ on this graph is referred to in the section *Experimental Confirmation.*

We should note that plots of x_1 against N are basically similar to the plots of R_N (except for small values of N, where the approximation which x_1 represents is invalid), as are those for various other possible measures, such as the maximum number of identifiers hashing to the same value. Note also that the algorithm GNU-cc1 sets $h_0 = n$ whereas all the other algorithms of that type set h_0=0.

This difference may be expensive in processing time, since in C implementations two traverses of the string are required. (In the experiments mentioned above this extra traverse amounted to 61 microseconds per identifier.) It may well be that in some applications the first (length finding) traverse has already been made for another purpose, nevertheless, plots of R_N for the two cases showed that, if anything, the extra complication (setting $h_0 = n$) has made the algorithm worse.

The key to understanding these graphs is to note that in Figure 3d the underlying structure is a straight line through (0, 1). Superimposed on this is some statistical noise. In addition there are sharp peaks at *k, 2k, 3k* etc. Inspection of Figure 3c confirms this interpretation. There are peaks at *k, 2k* etc., higher peaks at k^2, $2k^2$ etc. (There are still higher peaks at k^3, $2k^3$ but these had to be elided from the graph in order to keep the vertical scale reasonable.) Further inspection of Figure 3d suggests, and examination of plots drawn to a different scale confirms, that there are subsidiary peaks at approximately *k/2, 3k/2* etc. This enables us to interpret Figure 3b. The lowest strand corresponds to odd values of *N*. The next strand corresponds to odd multiples of 2 (i.e. *k/2, 3k/2* etc). The next to odd multiples of 4 and so on. Figure 3a has a basic similarity to Figure 3b, but in addition there are marked 'saw-teeth'. Notice that these saw-teeth have the property that if a steeply rising edge occurs from $x = i$ to $x = j$ then a similar rising edge occurs from $x = 2i$ to $x = 2j$.

Figure 5a is a plot of the number of identifiers hashed to each bucket by this algorithm for a 'good' value *viz*. $N = 1139$; Figures 5b & c are for a 'bad' value ($N = 893$) and for double that value ($N = 1786$). Although the distribution is not very uniform for $N = 1139$ it is certainly not as skewed as for $N = 893$. When $N = 1786$ we get basically the same pattern twice, with the second part overlaid by a certain amount of 'noise'. A rough explanation for this behaviour is not difficult to provide.

If N is a factor of $k.h_{n-1}$, then $H = c_n \ MOD \ N$, so that in these cases the behaviour of the algorithm is dependent only on the last character of the identifier.

If $N = rk$ then $H = c_n$ or $c_n + k$ or ... or $c_n + (r-1)k$.

If the values of h_{n-1} are uniformly randomly distributed then

$$f_{i,rk} \approx f_{i+k,rk} \approx ... \approx f_{i+(r-1)k,rk} \approx f_{i,k}/r \qquad (*)$$

hence

$R_{rk} \approx k\sum[f_{i,k}(f_{i,k} + r)]/W(W + rk)$ and so

$R_{rk} \approx R_k + k(r - 1)(1 - R_k)/(W + rk) < R_k$.

Since the observed values do not agree with this result, we may deduce that the values of h_{n-1} are **not** uniformly distributed. Indeed, for $n = 1$, this is obviously so, since then $h_{n-1} = h_0 = 0$ (in the cases that we are considering); for $n = 2$, inspection of the distribution of initial characters shows that they are not uniformly distributed. A further flaw in the above argument arises from the fact that k is not necessarily a factor of $r * k$ since integer overflow may have occurred in computation of the product. This situation arises quite early ($n = 5$) for $k = 613$.

Nonetheless, while the relations (*) above may not hold exactly because of these other effects, they hold sufficiently to produce the peaks at k, $2k$ etc. The higher peaks at k, k^2 etc arise from a similar consideration, namely that since

$H = (k^2.h_{n-2} + k.c_{n-1} + c_n) \ MOD \ N$

when N is a multiple of k^2 only the last two characters have significant effect.

It will be obvious from its derivation that R_N is a function of W. The values of R_N obtained also depend on the actual items being hashed. Consequently it would be dangerous to attempt any assessment of an algorithm on the basis of absolute values of R_N obtained. It is necessary, as has been done here, to examine the behaviour of R_N for a range of values of N.

EXPERIMENTAL CONFIRMATION

To test the deductions made about the behaviour of these algorithms a further experiment was conducted. The intention was to employ hashing in a situation in which one might typically find it used. The situation chosen was an algorithm for finding all the words which each occur exactly once in a given text. The Greek term *hapax legomenon* (plural *hapax legomena*) is used to denote such a word. The selected text (Thomas Hardy's novel *Far From The Madding Crowd*) contained 136 072 words in all, with 6281 *hapax legomena*. The hashing algorithms ACK, ETH-new, GNU-cpp, GNU-cc1, Icon and PCC were used, modified so that in each case 1787 buckets were used, and, in the case of GNU-cc1, h_0 was set to 0. The choice of 1787 for the number of buckets was made because it is an odd prime which occurs on a peak of the curve for algorithm PCC (see Figure 4). The results obtained are given in Table C.

Since in this application the speed of the hashing algorithm depends on both its intrinsic speed (as given by t in Table B) and also on the time spent in searching the chains resulting from collisions (whose lengths are a function of R_N in Table B) the ranking in Table C agrees with neither of those two measures alone, but contains them as weighted components. It does, however, confirm that GNU-cpp, though very simple, performs very well.

CONCLUSIONS

It seems to be the case that algorithms of the style

$$h_0 = 0; \quad h_i = k.h_{i-1} + c_i \quad \text{for} \quad 1 \le i \le n$$
$$H = h_n \ \text{MOD}\ N$$

can perform well, but k and N need to be selected with care. Although it may seem unlikely that anyone would choose one of the really bad combinations, the facts presented above indicate that far from optimal choices are made and persisted with. The experiments have shown that very small variations in N can produce large variations in the efficiency of the hash table lookup, and that the popular view, that choice of a prime number will automatically ensure a good result, is not well founded.

Values of k of the form 2^n are desirable in terms of speed, since then the multiplication can be done by shifting. Inspection of Figures 3 and 4 suggests that taking $k = 4$ and N odd has the advantage that the resulting routine will run very fast and yet a value of R_N can be obtained which is very close to that obtainable from 'better' but slower choices. A suitable value of N may be chosen by consulting Figure 4.

ACKNOWLEDGEMENTS

The authors wish to thank Warwick Heath, whose diligence unearthed the starting point of this investigation, Greg Ewing who shared in many of the discussions which helped shape it, and Professor Kevin Lee who settled a dispute about the plural of□ Απαξ λεγομενον.

REFERENCES

[1] TANENBAUM, A.S., VAN STAVEREN, H. AND STEVENSON, J.W. A practical toolkit for making portable compilers. *Commun. ACM* 26, 9(Sept. 1983), pp654-660

[2] KNUTH, D.E. *The Art Of Computer Programming*. Vol.3 *Sorting and Searching* ; Addison-Wesley; Reading; 1973; pp513-11

[3] ABRAMOWITZ, M. AND STEGUN, I (ed); *Handbook of Mathematical Functions;* Dover Publications; New York; 1965; pp227-241

TABLES and FIGURES