Performance in Practice of String Hashing Functions

M.V. Ramakrishna   Justin Zobel
Department of Computer Science, RMIT
{rama,jz}@cs.rmit.edu.au

Abstract

String hashing is a fundamental operation, used in countless applications where fast access to distinct strings is required. In this paper we describe a class of string hashing functions and explore its performance. In particular, using experiments with both small sets of keys and a large key set from a text database, we show that it is possible to achieve performance close to that theoretically predicted for hashing functions. We also consider criteria for choosing a hashing function and use them to compare our class of functions to other methods for string hashing. These results show that our class of hashing functions is reliable and efficient, and is therefore an appropriate choice for general-purpose hashing.

1 Introduction

String hashing is the process of reducing a string to a pseudo-random number in a specified range. It is a fundamental operation, used widely in applications where speed is critical. On a small scale, a hash table is often the basic data structure in applications such as symbol tables in compilers and account names in password files. Hashing is also used in applications such as spell checking and Bloom filters [15]. In databases, hashing is important, not just for indexing, but also for operations such as joins and inverted-file construction.

The performance of a hashing scheme depends primarily on two factors: the efficiency of the overflow-handling scheme and the behaviour of the hashing function. There has been much research addressing the problems of overflow and collisions. Hashing functions have received less attention, but analytically the behaviour of hashing is now well-understood [3, 7, 10, 11, 14]. However, in much of the work on hashing it is assumed that the keys are integers, while in practice keys are often strings of alphanumeric characters—an aspect of hashing that has attracted surprisingly little research. Some recent papers have examined specific string hashing functions [12, 13] but how these functions compare to the analytically-predicted performance of hashing is unknown.

Moreover, good choice of hashing function is crucial to efficiency. It is often assumed that for a given load factor access costs are independent of table size, but for a poor function this assumption breaks down. In comparison to a good hashing function, a badly designed function may give acceptable performance for a small application such as a symbol table but be much slower when used for a large database application such as a join.

In this paper we present a class of string hashing functions and demonstrate experimentally that the analytically-predicted performance can be achieved in practice by choosing hashing functions at random from this class; to our knowledge there has been no previous investigation of classes of string hashing functions. In these results performance is evaluated by two measures: the average number of probes during successful and unsuccessful search, and the largest number of probes during successful search, that is, the worst case. Our experimental results are based on sets of strings drawn from real data, including a set of over one million distinct words drawn from a text database. The results show that the class gives good average performance.

We also identify four properties that a class of string hashing functions should satisfy: uniformity, universality, applicability, and efficiency. We use these properties to motivate our class of string hashing functions and to compare it to other string hashing functions that have been proposed. These results show that functions in the class are, as well as reliable, faster than other good hashing functions. This class of functions is therefore a good choice for any application involving hashing of strings, including schemes such as hash joins and external hashing as well as the chained hashing used in this paper to explore function performance.

In Section 2 we describe our class of hashing functions. Analysis of hashing schemes is reviewed

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1–4, 1997.

in Section 3. In Section 4 we describe our test data and experimental results, considering both average-case and worst-case search lengths. Other string hashing functions are discussed in Section 5.

2 Classes of hashing functions

In this section we describe a class of string hashing functions. First we outline our notation.

String hashing functions can be represented in the following generic form, in which $s = c_1 \ldots c_m$ is a string of m characters, v is a seed, and h_i is an intermediate hash value after examination of i characters.

```
$h(s,v) =$
    set $h_0 \leftarrow \mathit{init}(v)$
    for each character $c_i$ in $s$,
        set $h_i \leftarrow \mathit{step}(i, h_{i-1}, c_i)$
    return $h = \mathit{final}(h_m, v)$
```

That is, the hash value of s is computed as follows. Some function *init* is applied to v to yield the initial h_0. At each step h_i is a function *step* of i, the hash value computed so far, and the current character. The hash value returned is a function *final* of v and the internal hash value h_m. Defining *init*, *step*, and *final* describes a string hashing function. For example, we might define

$$\begin{aligned} \mathit{init}(v) &= 0 \\ \mathit{step}(i,h,c) &= h + c \\ \mathit{final}(h,v) &= h \end{aligned}$$

yielding a simple (rather uninteresting) hashing function in which the hash value is the sum of the ASCII values in the given string.

Hash values must be truncated in some way to give values in the range $0 \ldots T-1$, where T is the table size. In general the only practical mechanism is to take h modulo T (remainder of h after division by T) but, for values of T of the form 2^b with integer b, bitwise AND can be used.

Operations that might be used in a hashing function include addition ($+$), multiplication ($\times$), bitwise AND ($\vee$), bitwise OR ($\wedge$), bitwise exclusive OR ($\oplus$), modulo ($||$), left-shift of value h by b bits ($\mathcal{L}_b(h)$), and right-shift of value h by b bits ($\mathcal{R}_b(h)$). On most architectures today modulo is implemented in software; multiplication, although usually in hardware, is relatively slow; while the other operations are typically single-cycle instructions. We assume that characters are represented in some integer code such as ASCII.

We contend that, to be useful for general-purpose hashing, a class of hashing functions should satisfy the following properties.

Uniformity. If a hashing function is uniform then the probability of an arbitrary key hashing to a given slot is $1/T$ for table size T, independent of the hash values of other keys. In practical terms, uniformity means that for a given *load factor* (ratio of keys to slots) average access time is roughly constant, regardless of table size.

Universality. A class of hashing functions H is universal if, for a given table size T and any pair of valid keys s_1 and s_2, the number of hashing functions $h \in H$ such that $h(s_1) = h(s_2)$ is less than or equal to $|H|/T$ [2]. That is, for a randomly-chosen hashing function the probability that s_1 and s_2 hash to the same value is less than or equal to $1/T$.

In practice universality means that, with high probability, a randomly-chosen hashing function will perform well. For any hashing function it is true that there exist sets of keys that all hash to the same value—and no hashing function is invulnerable to a deliberate attempt to identify such a set of keys. However, if a class of hashing functions is universal and the functions in the class are uniform then it is guaranteed that the class cannot be subjected to such attack. If for some hashing function h and set S of keys every key $s \in S$ is such that $h(s) = k$ for some k, it is still true that for another randomly-chosen function h' the set of hash values $h'(s)$ will be uniformly distributed.

It is somewhat difficult to test universality in practice; such a test would require hashing every pair of keys for every possible seed value and table size. However, by subjecting the class to attack of the kind outlined above—actively searching for keys that hash to the same value—we can obtain a strong indication that the class is indeed universal.

Applicability. At a more pragmatic level, hashing functions should be applicable in all circumstances where hashing might be used. A function that is limited to a few table sizes, can only hash strings of a certain length, or cannot accept seeds (thus allowing, for example, double hashing) is not as valuable as functions without such restrictions.

Efficiency. The primary advantage of hashing as an access method is its speed: given a data set of n keys and a table of $O(n)$ size search time is O(1), assuming a hashing function with time complexity $O(1)$. Hashing functions should also be small; in many applications there is little advantage to a function that is as large as the key set.

In practice, constant factors can be important. For example, in some applications it is possible for search in a array, although $O(\log n)$, to be as fast as search in a hash table. Consider an application in which the keys are long strings. During search of an array, all that is required at each element is inspection of the first few characters of the key (up to a mismatch), then a full comparison when the correct element is found. During hashing, the key must be completely inspected at least twice, once to form the hash value and at least once to check the key in the table. A slow hashing function, with complex operations for each character, could well be unacceptable.

Other valuable properties are perfection, where the hashing function is collision-free, and order-preservation, where the sort order of the hash values is the same as the sort-order of the original keys [1, 4, 16]. Both are valuable in specific applications; perfect hashing functions can be used for lookup in static tables, for example, because it may then not be necessary to store the keys. However, such functions require prior knowledge of the complete set of keys to be hashed. We do not consider perfect or order-preserving hashing functions in this paper.

We now define our class of string hashing functions. To obtain a class of hashing functions that meets the criteria above, we wish to use efficient primitive operators such as addition and exclusive OR; to use as few as possible of these operators; to allow the function to generate large hash values; and to design the function to scramble the input bits as thoroughly as possible, without losing the contribution of any characters. Thus it is essential to use some mechanism such as left shift to make use of higher-order bits, while operators such as bitwise AND should be avoided because they tend to erase information. Based on these principles we experimented with many combinations of primitive operators, and as a result propose the *shift-add-xor* class of hashing functions, in which the components are defined by

$$\begin{aligned} \mathit{init}(v) &= v \\ \mathit{step}(i,h,c) &= h \oplus \big(\, \mathcal{L}_L(h) + \mathcal{R}_R(h) + c \,\big) \\ \mathit{final}(h,v) &= h \| T \end{aligned}$$

in which the modulo operation in the final step can be replaced by bitwise-AND for suitable T values. As we discuss below, this function was the simplest we could identify that had the required properties. Functions of this general form are not new, but to our knowledge they have not previously been analysed with respect to the theoretical behaviour of hashing functions.

Uniformity and universality are investigated in Section 4; for now we simply note that each seed gives a new function and hence we have defined a class of hashing functions.

Given appropriate choice of shift magnitudes L and R good use is made of the 32-bit space, providing greater likelihood of a uniform distribution of hash values. For example, with 5-character keys and $L \geq 7$ it is possible to obtain any 32-bit string using this class of functions. We used $L = 5$ and $R = 2$ in our experiments, but found that variation in these values made little difference for $4 \leq L \leq 7$ (so that only a few characters are required to yield a large hash value) and $1 \leq R \leq 3$ (so that the contribution of the first characters is not diminished); note that the character set was ASCII, that is, 7-bit values. We conclude that the class is widely applicable.

The class is fairly efficient. There is no use of slow operations such as modulo or multiplication (other than the necessary final use of modulo to reduce the hash value to the table size) and only five operations per character—one exclusive OR, two shifts, and two additions—each of which on our machines require only a single instruction cycle. It is possible that there exists a simpler effective hashing function, but it cannot be obtained by simplifying shift-add-xor. Considering the possible simplifications: the left-shift is required to obtain 32-bit values; the right-shift is required for uniformity, because, we suspect, the majority of occurrences of letters in English, including all six vowels, have a rightmost 1-bit; and, as we discuss below, the exclusive OR is required for universality. Efficiency is considered further in Section 5.

Note that we do not require that table size T be prime, or be carefully chosen in any way; hashing functions should be effective for all table sizes.

Some readers may be curious as to why we chose to define *step* as above rather than as

$$\mathit{step}(i,h,c) \quad = \quad h \oplus \big(\, \mathcal{L}_L(h) \oplus \mathcal{R}_R(h) \oplus c \,\big)$$

given the belief that exclusive OR is appropriate for hashing. This shift-xor-xor class is uniform but, apparently, not universal. The reasons are not entirely clear to us, but it seems that a mix of addition and exclusive OR is required; in particular, addition appears to be valuable because it propagates change between bits, and leads to a more even distribution of 0 and 1 at each position.

There are essentially two methods for hashing strings. One is to directly reduce the string to a string of bits, as in the shift-add-xor class. Another is to convert the string to a number, then apply an integer hashing function such as

$$\mathit{final}(h, v) \quad = \quad \big((v \times h) \| p\big) \| T$$

where p is a large prime. We would expect such functions to be well-behaved, but the operations required in the conversion and hashing make it unlikely that they would be faster than shift-add-xor. For example, consider Cormen, Leiserson, and Rivest [2] and Sedgewick [17], which are two of the better-known recent algorithms texts. Cormen, Leiserson, and Rivest [2] suggest that strings be converted to numbers through radix conversion. For alphanumeric strings, implicitly of base 62, radix conversion requires up to two comparisons, a subtraction, and a multiplication for each character, with possibly further operations because of overflow. For ASCII characters radix conversion is rather simpler, involving only a left-shift of 7 places—multiplication by 128—but such conversion can lead to the contribution of the first characters in a string being lost as they are shifted out to the left. Thus the technique of regarding strings as numbers and using numerical hashing is inappropriate unless arbitrarily large numbers can be manipulated efficiently. Sedgewick [17] suggests a method that avoids overflow by use of a modulo operation at each step, which is considerably more expensive to evaluate.

Our hypothesis, then, is that by choosing functions at random from the shift-add-xor class of string hashing functions—that is, by making a random selection of seed—we can in practice obtain the analytically predicted performance of hashing schemes. Prediction of performance is reviewed in the next section.

3 Predicted behaviour of hashing

Hashing techniques are usually analysed under the assumption that the hash values are uniformly distributed. Consider a set of n keys mapped into an address range of T values. Given a key s and a hashing function h that maps the key into this range, the probability that the key hashes to a particular address is $1/T$ and is independent of the outcome of hashing other keys. There are T^n ways in which n keys can be distributed among the T addresses, that is, there are T^n functions that map the given set of keys into the table. It is assumed that each of these distributions is equally likely when the n keys are hashed into T slots.

The analytically-predicted performance of a class of hashing functions corresponds to the expected performance of a randomly-chosen function from the set of T^n functions. It is interesting to consider both average-case and worst-case behaviour. In the average case, behaviour is measured by the average length of the probe sequence (that is, the average number of accesses) for successful and unsuccessful search. Analytical results for the average case in a chained hash table are given by Knuth [8, page 535].

The worst case for hashing occurs when all the keys hash to the same address and the search length is $O(n)$. Knuth [8, page 540] expressed fear of this possibility by concluding that "hashing would be inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake". However, Gonnet [5] proved that such fears of hashing are baseless, since the probability of the worst case is, in his words, ridiculously small.

Gonnet proposed a measure for the worst case of hashing based on the length of the longest probe sequence, or *llps*. Out of all the keys stored in the hash table, one has the maximum successful search length. Gonnet proposed that the expected value of llps is a better measure of the worst case of hashing than is the (extraordinarily improbable) worst case of llps, and demonstrated theoretically that llps is very narrowly distributed with the expected value being quite small, that is, not dramatically greater than would be given by dividing the keys evenly amongst the buckets. Larson extended these results for the general case of bucket size greater than 1 [9].

We now use these analytical results, for both average-case and worst-case behaviour of a class of ideal hashing functions, as a yardstick for evaluating the behaviour in practice of classes of string hashing functions.

4 Experimental results

Our hypothesis is that, by choosing hashing functions at random from the shift-add-xor class of hashing functions, the analytically-predicted performance of hashing schemes can be achieved in practice. To support the hypothesis, in this section we experimentally evaluate the shift-add-xor class of string hashing functions on real data sets.

Exhaustively checking whether a class of hashing functions is indeed uniform would require evaluating the function over all potential key sets for all seeds. This or any close approximation to it is clearly impractical, but by applying the class to a selection of data sets with a reasonable number of seeds we can be highly confident that the observed behaviour is a good approximation to the behaviour over the whole class.

For these experiments we had several sets of keys available to us. We used these to explore the performance of the hashing functions discussed in this paper. The reported results are based on the following key sets. (However, results for all of the key sets were similar.) One was NAMES, a file of 31,918 distinct surnames extracted from Internet news articles and hand-edited to remove errors and nonsense [18].[1] Another was TREC, a file of 1,073,726 distinct words (that is, contiguous alphabetic strings) extracted from the first 3 gigabytes of the TREC data [6]; this data contains the full-text of newspaper articles, abstracts, and scientific journals. In our experiments we have focused on certain table sizes and load factors, to allow comparison with previously published analytical results, and thus did not usually require the full data sets. Instead we used subsets of the data of the required size: ten random subsets of 1000 strings each, from each of TREC and NAMES; the lexicographically first 1000 strings from each of TREC and NAMES; a file, FIVES, of the first 1000 distinct strings of exactly five characters (that is, "aaaaa", "aaaab", and so on); and a file, SEVIF, of the strings from FIVES reversed. These last four files are pathological cases that should help to expose flaws in weak hashing functions.

In these experiments we have focused on hash tables with separate chaining, which—with their tolerance to overflows and similarity to dynamic-table schemes such as linear hashing and extensible hashing—we consider to be most typical of hash tables in practical use. However, the results are independent of the hash table organisation: they demonstrate properties of the class of hashing function that apply regardless of how it is used, whether for internal or external hashing, to slots of size 1 or buckets of many keys each, or to applications such as hash joins.

Average-case search length

We first investigated average search lengths for successful and unsuccessful search. Results are shown in Table 1. The "actual" results are an average over 10,000 randomly-selected hashing functions (equivalently, seeds), based on one set of TREC keys; the "±" figure is one standard deviation. In these results the number of keys was held at 1000 and the table size varied to give a load factor. For example, with a load factor of 70% the table size was $\lceil \frac{1000}{70\%} \rceil = 1429$. The "predicted" results are quoted from Knuth [8, page 535]. As can be seen, the correspondence is extremely good, thus confirming our hypothesis that functions in the class shift-add-xor generate uniform hash addresses. Almost identical results—usually to within 0.01—were observed with the other data files, including the four "pathological" data files. We also tried other table sizes and key set sizes, including table sizes such as powers of 2 that might lead to poor behaviour, but again similar behaviour was observed. Note that we have not reported figures for larger bucket sizes; changing bucket size does not change the distribution or the properties of the hashing function.

By way of comparison, consider the class of hashing functions given by

$$\begin{aligned} \mathit{init}(v) &= 0 \\ \mathit{step}(i,h,c) &= \mathcal{L}_1(h) + c \\ \mathit{final}(h,v) &= h \| T \end{aligned}$$

This function is like that used in several compilers, as reported by McKenzie, Harries, and Bell [12]. For a load factor of 90%, on the twenty randomised data files average successful search length was 1.701, already significantly greater than the prediction of 1.450, while on SEVIF it was 5.110 and on FIVES it was 9.358.

It is also interesting to consider performance on a large data set, the more realistic case for a hashing function to be used in a database system. For the full set of TREC keys, 1000 randomly-chosen hashing functions, and a load factor of 90%, shift-add-xor gave an average successful search length of 1.459 and the average search length was 1.310—essentially identical to performance with a small set of keys. With the simple function above, however, average successful search length was 19.103; increasing the value of the left shift to 4 decreases this value, to 4.669, a figure that is however still unacceptable. From this experiment and similar experiments with other large sets of keys (such as the first 1,000,000 five-character strings) we have observed that with a poorly-chosen hashing func-

[1] This file is available by ftp from
goanna.cs.rmit.edu.au
in the file
pub/rmit/fnetik/data/Surnames.Z

	40%	60%	70%	80%	90%
Predicted successful	1.200	1.300	1.350	1.400	1.450
Actual successful	1.200±0.014	1.299±0.017	1.350±0.019	1.400±0.020	1.450±0.021
Predicted unsuccessful	1.070	1.149	1.197	1.249	1.307
Actual unsuccessful	1.070±0.004	1.148±0.006	1.196±0.007	1.249±0.008	1.307±0.009

Table 1: Average search length, successful and unsuccessful for 1000 keys at load factors from 20% to 90%, averaged over 10,000 seeds (± one standard deviation). The keys are extracted from the TREC data.

tion performance can markedly deteriorate as the number of keys increases. However, a good hashing function such as a member of shift-add-xor will indeed give the theoretically-predicted performance.

Worst-case search length

Experimental results for the expected length of the longest probe sequence, or llps, are shown in Table 2, from the same experiments reported in Table 1. The "predicted" results are quoted from Gonnet [5]. As can be seen llps values vary significantly between runs, as indicated by the high standard deviation. For a load factor of 60%, the greatest llps observed in the 10,000 runs was 8; for load factors of 70%, 80%, and 90% the greatest llps was 9. The llps values varied somewhat between data files—for example, for a load factor of 90% and the files drawn from NAMES and TREC the minimum average value of llps was 5.257 and the maximum was 5.332. However, all of these values are, within the error indicated by the standard deviation, close to the analytically-predicted value. For FIVES, average llps was 3.034.

We decided to examine in detail the distribution of llps values, by hashing the strings in one data set with 1,000,000 randomly-selected hashing functions. The results are shown in Figure 1. As predicted by the analysis, the distribution of experimental llps values is extremely narrow—even with a load factor of 90%, over 95% of the llps values are 4, 5, or 6; the largest observed value was 12, which occurred only once in the 4,000,000 experiments.

Pushing this experiment further, we chose a random set of 20 keys from NAMES, a table size of 20, and measured llps for the 2^{30} hashing functions given by the seeds between 1 and 2^{30}. The worst llps was 11, with only 9 in over one billion occurrences. That is, for even such a small table exhaustive search of the class failed to find a hashing function that maps all keys to the same value. Average llps was 3.231.

For the full set of TREC keys, the distribution of llps values is even narrower. With 1000 randomly-chosen hashing functions and with a load average of 90%, the average llps was 8.900, with a minimum of 8 and a maximum of 11. (Note that llps is expected to rise slowly as table size is increased; this is not an indicator of poor performance.) Interestingly, our experiments indicate that llps is a better tool than average search length for discriminating between hashing functions, particularly on large key sets. For example, on the same data the hashing function given by simplifying the *step* operation in shift-add-xor to

$$step(i, h, c) \quad = \quad h \oplus \left(\mathcal{L}_L(h) + c \right)$$

has reasonable average successful search length but average llps—the worst-case successful search length—markedly deteriorates, to 25.886.

Note that the llps values quoted in Table 2 are not a lower bound—it is quite possible for a hashing function to have better worst-case performance for a given data set. In particular, perfect hashing functions, which are constructed with respect to the set of keys to be hashed, have by definition an llps of 1. The weakness of such functions is their inefficiency for dynamic sets of keys and the unpredictable behaviour for an arbitrary key set.

Universality

Although it is not possible to conclusively demonstrate that a class of hashing functions is universal, there is evidence that can indicate whether universality holds. The method we have used is deliberate attack: for some hashing function and table size, find a set of strings that hash to the same value; then for that set of strings explore llps and average search lengths. A significant increase in llps indicates that some strings are being hashed to the same value for more seeds than would be expected for a universal class of hashing functions.

To use this approach to provide evidence for universality we used the full TREC key set, assumed a loading factor of 90% and a table size of 1111, randomly chose a hashing function, then searched for

	60%	70%	80%	90%
Predicted	4.333	4.636	4.947	5.242
Actual	4.556±0.644	4.797±0.677	5.069±0.679	5.306±0.688

Table 2: Length of the longest probe sequence (llps) for 1000 keys at load factors from 60% to 90%, averaged over 10,000 seeds (± one standard deviation). The keys are extracted from the TREC data.

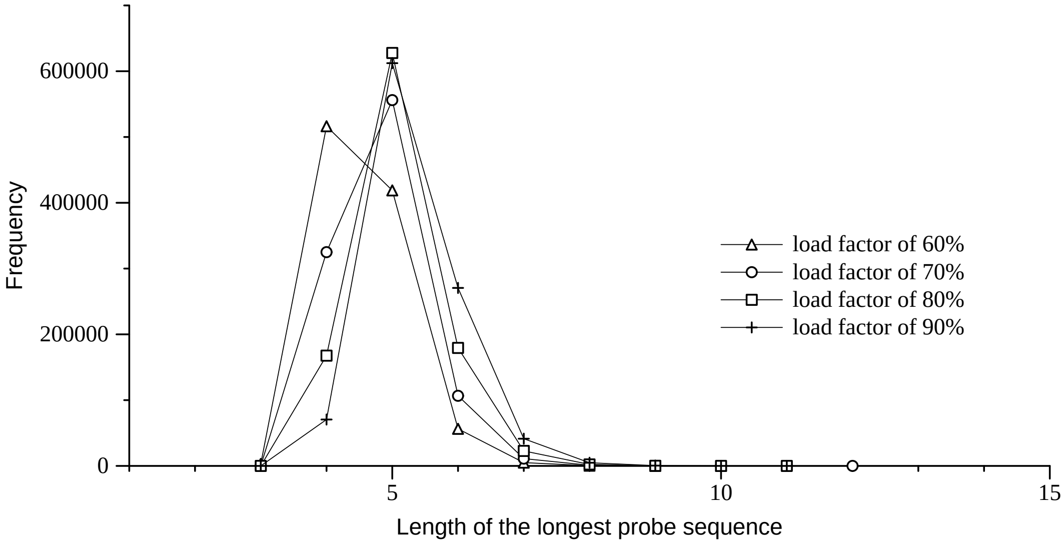

Figure 1: Distribution of the length of the longest probe sequence (llps) for 1000 keys and 1,000,000 randomly-selected seeds. The keys are extracted from the TREC data.

any set of 1000 keys with the same hash value. We then attacked the shift-add-xor class by choosing 1,000,000 random seeds and examined the distribution of llps values. After this attack the average llps was 5.307—almost identical to the value in Table 2—and, aside from one occurrence of an llps of 15 and one of an llps of 12, the values were between 4 and 10. That is, behaviour was virtually indistinguishable from that of a random set of keys.

This process of attack was a key tool in our evaluation of other hashing functions. For example, for the shift-xor-xor class defined with

$$\mathit{step}(i,h,c) \quad = \quad h\oplus\big(\mathcal{L}_L(h)\oplus\mathcal{R}_R(h)\oplus c\big)$$

attack increases average llps over 1000 seeds, to 6.229. For the class defined with

$$\mathit{step}(i,h,c) \quad = \quad \mathcal{L}_L(h)+\mathcal{R}_R(h)+c$$

average llps is increased to 41.198, with an average successful search length of 5.491.

To survive such an attack, a class of functions must be large. For example, the class defined by

$$\begin{aligned}\mathit{init}(v) &= v\\ \mathit{step}(i,h,c) &= \mathcal{L}_7(h)+c\\ \mathit{final}(h,v) &= h\|T\end{aligned}$$

only has, in effect, T distinct members for a table size T that is a power of 2.

5 Other string hashing functions

In addition to string hashing functions proposed in the literature, not surprisingly many different functions are to be found embodied in software. We now consider some of these functions.

As discussed above, some algorithms texts describe a form of multiplicative method, which are two-stage methods in which the string is reduced to a number before processing with a hashing function for integers. One form of multiplicative method is as follows.

$$\begin{aligned}\mathit{init}(v) &= 0\\ \mathit{step}(i,h,c) &= h\times r+c\\ \mathit{final}(h,v) &= \big((v\times h)\|p\big)\|T\end{aligned}$$

where p is a large prime and r is a radix. In a variation of this form, an array P of distinct large primes can be used as follows.

$$\begin{aligned}\mathit{init}(v) &= 0\\ \mathit{step}(i,h,c) &= h+P_i\times c\\ \mathit{final}(h,v) &= \big((v\times h)\|p\big)\|T\end{aligned}$$

We tested several functions of this kind using the same methodology as in Section 4. These approaches are uniform (provided that radix r is not a power of 2); probably universal, as they are resistant to adversarial attack; and widely applicable. However, Sedgewick's function [17] does not resist attack quite as well as other functions on the full set of TREC strings; average llps is 10.334, a significant increase.

Moreover, these functions are relatively slow. On a Sun SPARC 20, functions from the shift-add-xor class can process just over 1000 strings can be hashed per millisecond. In contrast, the multiplicative methods processed under 300 strings per millisecond and Sedgewick's method (which uses a modulo for each character) processed under 150 strings per millisecond. We would expect similar relative performance on other current architectures.

Two recent papers concern string hashing functions. Pearson [13] proposed an algorithm that can be defined as

$$\begin{aligned} \mathit{init}(v) &= 0 \\ \mathit{step}(i,h,c) &= A_{h\oplus c} \\ \mathit{final}(h,v) &= h \end{aligned}$$

in which A is an array of the 256 distinct 8-bit values, randomly permuted, and $A_{h\oplus c}$ denotes the $h\oplus c$'th value in A. This algorithm computes 8-bit hash values only, albeit quickly. Pearson also gives an extension to 16-bit hash values, which is somewhat slower. The array A is in effect the seed, since different permutations yield different hashing functions. However, this function is of limited value in practice, since it is expensive to store each seed, and the function is only applicable to limited table sizes. As a generalisation of Pearson's function we tested the class

$$\begin{aligned} \mathit{init}(v) &= v \\ \mathit{step}(i,h,c) &= h\oplus \left(\mathcal{L}_L(h) + A_{(h\oplus c)\|256} \right) \\ \mathit{final}(h,v) &= h\|T \end{aligned}$$

This class is uniform—experimental results are almost identical to those for shift-add-xor—resistant to attack, and at around 800 keys per millisecond is only slightly slower than shift-add-xor. We have found no simplification of this algorithm that preserves uniformity and universality.

The other recent paper on string hashing functions is a survey of their use in software, by McKenzie, Harries, and Bell [12]. Like the function described by Pearson, these functions are not designed to accept seeds (and thus do not form classes) and, with respect to our criteria, are not particularly interesting. Nor are they distinct from the classes we have already discussed; most are variants of simple radix or shift methods.

A more interesting class of hashing functions is defined by

$$\begin{aligned} \mathit{init}(v) &= v \\ \mathit{step}(i,h,c) &= c\oplus \big(\mathcal{L}_L(h)\vee \\ &\qquad (\mathcal{R}_{32-L}(h) \wedge \mathit{MASK}) \big) \\ \mathit{final}(h,v) &= h\|T \end{aligned}$$

in which *step* is a left-rotation of h by L bits XOR'ed with c and MASK is 2^L-1, that is, L one-bits. This function, which is similar to a method attributed by Knuth to Knott [8, page 412] and is embodied in the `ispell` spelling checking utility, has almost exactly the same cost as shift-add-xor and is thus about the same speed, but is slightly vulnerable to attack, with an average llps of 6.064.

6 Conclusions

Analytically, the behaviour of hashing schemes is well understood. In this paper we have presented criteria by which we believe practical hashing functions should be evaluated—uniformity, universality, applicability, and efficiency. We developed a class of *shift-add-xor* string hashing functions and experimentally showed that, by choosing hashing functions at random from this class, the analytically-predicted performance can be achieved in practice. We have also shown that the class is likely to be universal, as it is resistant to one method of adversarial attack. Moreover, the functions from the class are computationally efficient, processing more keys per unit time than other good string hashing functions, and, as shown in our experiments with the distinct words in TREC, are effective even for a large key sets such as strings in a database.

The shift-add-xor class of functions is thus an appropriate choice for practical applications. Our results answer an important question posed by all users of hashing, namely, what function should be used for hashing strings. The answer is, make a random choice from this class; with high probability it will work well, and will be at least as efficient as other effective hash functions.

The worst-case performance results for the shift-add-xor class of string hashing functions are of particular interest. We have provided experimental evidence—including, for one set of strings, exhaustive search amongst one billion hashing functions—confirming the theoretical prediction that the length of the longest probe sequence is narrowly distributed. To our knowledge these are the first experiments testing this prediction. These results are also a further confirmation that, with an appropriately chosen class of hashing functions, hashing is indeed safe in practice—the likelihood of the theoretical worse-case of many keys hashing to the same value is extraordinarily low.

Acknowledgements

We thank Evan Harris for suggesting several hashing functions. We also thank Kotagiri Ramamohanarao. This work was supported by the Australian Research Council.

References

[1] G.V. Cormack, R.N.S. Horspool and M. Kaiserwerth. Practical perfect hashing. *Computer Journal*, Volume 28, Number 1, pages 54–55, February 1985.

[2] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Massachusetts, 1990.

[3] R.F. Deutscher, P.G. Sorenson and J.P. Tremblay. Distribution dependent hashing functions and their characteristics. In *Proc. ACM-SIGMOD International Conference on the Management of Data*, pages 224–236, 1975.

[4] E.A. Fox, Q.F. Chen, A.M. Daoud and L.S. Heath. Order-preserving minimal hash functions and information retrieval. *ACM Transactions on Information Systems*, Volume 9, Number 3, pages 281–308, 1991.

[5] G. Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM*, Volume 28, Number 2, pages 289–304, 1981.

[6] D.K. Harman. Overview of the first Text Retrieval Conference. In D.K. Harman (editor), *Proc. TREC Text Retrieval Conference*, pages 1–20, Washington, November 1992. National Institute of Standards Special Publication 500-207.

[7] G.D. Knott. Hashing functions. *Computer Journal*, Volume 18, Number 3, pages 265–278, 1975.

[8] D.E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition.* Addison-Wesley, Massachusetts, 1973.

[9] P. Larson. Expected worst-case performance of hash files. *Computer Journal*, Volume 25, Number 3, pages 347–352, 1982.

[10] V.Y. Lum. General performance analysis of key-to-address transformations methods using an abstract file concept. *Communications of the ACM*, Volume 16, Number 10, pages 603–612, 1973.

[11] V.Y. Lum, P.S.T. Yuen and M. Dodd. Key-to-address transform techniques: A fundamental performance study on large existing files. *Communications of the ACM*, Volume 14, Number 4, pages 228–239, 1971.

[12] B.J. McKenzie, R. Harris and T. Bell. Selecting a hashing algorithm. *Software—Practice and Experience*, Volume 20, Number 2, pages 209–224, 1990.

[13] P.K. Pearson. Fast hashing of variable-length text strings. *Communications of the ACM*, Volume 33, Number 6, pages 677–680, 1990.

[14] M.V. Ramakrishna. Hashing in practice, analysis of hashing and universal hashing. In *Proc. ACM-SIGMOD International Conference on the Management of Data*, pages 191–199, 1988.

[15] M.V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, Volume 32, Number 10, pages 1237–1239, 1989.

[16] M.V. Ramakrishna and P.A. Larson. File organization using composite perfect hashing. *ACM Transactions on Database Systems*, Volume 14, Number 2, pages 231–263, 1989.

[17] R. Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, Massachusetts, second edition, 1990.

[18] J. Zobel and P. Dart. Phonetic string matching: Lessons from information retrieval. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 166–173, Zurich, Switzerland, August 1996.