4-12-2012

The exponential hash function

Chaouki T. Abdallah

Bradley J. Smith

Gregory L. Heileman

The Exponential Hash Function

Bradley J. Smith, Gregory L. Heileman and Chaouki Abdallah
University of New Mexico, Albuquerque, New Mexico

In this paper an efficient open address hash function called exponential hashing is developed. The motivation for this hash function resulted from our ongoing efforts to apply dynamical systems theory to the study of hashing; however, the analysis conducted in this paper is primarily based on traditional number theory. Proofs of optimal table parameter choices are provided for a number of hash functions. We also demonstrate experimentally that exponential hashing essentially matches the performance of a widely-used optimal double hash function for uniform data distributions, and performs significantly better for nonuniform data distributions. We show that exponential hashing exhibits a higher integer Lyapunov exponent and entropy than double hashing for initial data probes, which offers one explanation for its improved performance on nonuniform data distributions.

Categories and Subject Descriptors: E.1 [**Data Structures**]: tables; E.2 [**Data Storage Representation**]: hash-table representations; H.3.3 [**Information Storage and Retrieval**]: Information Storage and Retrieval

General Terms: Algorithms

Additional Key Words and Phrases: Chaos, dynamic dictionary ADT, dynamical systems theory, exponential hashing, Lyapunov exponent, number theory

1. INTRODUCTION

The hash table is a well-known data structure used to maintain dynamic dictionaries. A *dynamic dictionary* is defined as a collection of data items that can be accessed according to the following operations:

(1) *Search*(k, S). Returns the data item with key k in dynamic dictionary S.

(2) *Insert*(x, S). Adds data item x to dynamic dictionary S.

(3) *Delete*(k, S). Removes the data item with key k from dynamic dictionary S.

Dynamic dictionaries are ubiquitous in computing applications; they are widely used in databases, operating systems, compilers, and a range of business and scientific applications. The hash table data structure consists of an array T whose N slots are used to store the collection of data items. When implementing the above operations, an index is computed from the key value using an *ordinary hash function* h, which performs the mapping

$$h : U \rightarrow \{0, 1, \ldots, N-1\},$$

where U denotes the set of all possible key values (i.e., the universe of keys). Thus, $h(k)$ denotes the index, or *hash value*, computed by h when it is supplied with key $k \in U$. Furthermore, one says that k *hashes* to slot $T[h(k)]$ in hash table T.

This research was supported by The Boeing Company under contract W-300445.
Authors' addresses: Department of Electrical & Computer Engineering, University of New Mexico, Albuquerque, NM 87131-1356

Since $|U|$ is generally much larger than N, h is unlikely to be a one-to-one mapping. In other words, it is very probable that for two keys k_i and k_j, where $i \neq j$, $h(k_i) = h(k_j)$. This situation, where two different keys hash to the same slot, is referred to as a *collision*. Since two items cannot be stored at the same slot in a hash table, the *Insert* operation must resolve collisions by relocating an item in such a way that it can be found by subsequent *Search* and *Delete* operations.

One method of resolving collisions, termed *open addressing* by Peterson [1957], involves computing a sequence of hash slots rather than a single hash value. This sequence is successively examined, or *probed*, until an empty hash table slot is found in the case of an *Insert* operation, or the desired item is found in the case of *Search* or *Delete* operations. In open addressing the ordinary hash function discussed above is modified so that it uses both a key, as well as a probe number when computing a hash value. This additional information is used to construct the probe sequence. That is, in open addressing, hash functions perform the mapping

$$H : U \times \{0, 1, \ldots, N-1\} \rightarrow \{0, 1, \ldots, N-1\}$$

and produce the *probe sequence* $< H(k,0), H(k,1), H(k,2), \ldots >$. Because the hash table contains N slots, there can be at most N unique elements in a probe sequence. A *full length probe sequence* is defined to be a probe sequence that visits all N table entries using only N probes.

Much of the existing research on hash table implementations of dynamic dictionaries is based on statistical analyses, typically focusing on average-case performance and uniformly distributed data. The work presented here is distinguished from much of the previous research on hashing in that we treat open address hash functions as iterators; which allows us to employ tools from the field of nonlinear dynamical systems.

The remainder of this paper is organized as follows. In Section 2, a basic theoretical analysis is given for two of the most popular open address hashing algorithms, linear probing and double hashing. In the next section we introduce the Lyapunov exponent, a method used to detect chaos. We then discuss the meaning of the Lyapunov exponent in the integer domain, and its importance in analyzing probing behavior. In particular, after pointing out the relationship between good hash functions and chaotic iterators, we develop a technique for measuring chaos in hash functions. The analysis of hashing from this perspective motivated the development of a new hash function called exponential hashing, which we present in Section 4, along with theoretical and empirical comparative analyses with double hashing. Our experimentation shows that exponential hashing performs nearly as well as double hashing for uniformly distributed data, and performs substantially better than double hashing on non-uniformly distributed data.

2. OPEN ADDRESSING TECHNIQUES

Knuth [1973] notes that the desirable properties of an open address hash function include:

—Efficient hash function evaluation time.

—A long probe sequence to accommodate tables near capacity.

—Different probe sequences for each data item to avoid primary and secondary clustering, as defined below.

—Uniform distribution over the entire hash table for both the initial and subsequent probes. This property is widely known as the *uniform hashing* property [Heileman 1996].

A look at two common open addressing strategies, linear probing and double hashing, lays the theoretical groundwork for our study of exponential hashing.

2.1 Linear Probing

Linear probing is easily analyzed from a theoretical point of view, and forms the basis for the double hashing method described next. The linear probe function can be written as:

$$H(k,i) = (h(k) + ic) \bmod N, \tag{1}$$

where k is the key, $h(k)$ is an ordinary hash function that maps the key space to an initial location in the table, i is the probe index, N is the table size, and c is a constant. For the simplest case, with $c = 1$, this hash function will simply probe the sequential locations $< h(k) \bmod N, (h(k)+1) \bmod N, (h(k)+2) \bmod N, \ldots >$. For larger values of c, the function will probe table items at fixed distances of c modulo N in the table. Analysis of the best choice for c to achieve a full length probe sequence for a given table size is straightforward. Since the value $h(k)$ is fixed for a given key, substituting $x_0 = h(k)$, and letting $x_i = H(k,i)$ in equation (1) yields

$$x_i \equiv x_0 + ic (\mathrm{mod} N). \tag{2}$$

Since both addition and subtraction are closed modulo N, these terms can be rearranged to obtain

$$ic \equiv x_i - x_0 (\mathrm{mod} N).$$

From this is follows that a full length probe sequence is created if a unique value of $x_i - x_0 (\mathrm{mod} N)$ is obtained for each value of $i = 0, 1, \ldots N-1$. This condition is satisfied if c has a unique multiplicative inverse modulo N. A basic result from elementary number theory states that c has a unique multiplicative inverse modulo N, denoted c^{-1}, if and only if c and N are relatively prime. Therefore, an optimal strategy for arbitrary table size N is to choose c relatively prime to N, which guarantees full length probe sequences.

Simple empirical tests reveal the limitations of linear probing. For a given constant c, all initial hash values $x_0 = h(k)$ will produce the same probe sequence. This problem is known as *primary clustering*. Linear probing also leads to *secondary clustering* where for any two keys k_1 and k_2 with $k_1 \neq k_2$ and $h(k_1) = h(k_2)$, the sequences $< H(k_1,1), H(k_1,2), \ldots >$ and $< H(k_2,1), H(k_2,2), \ldots >$ will be identical. Both types of clustering are obviously undesirable.

2.2 Double Hashing

Double hashing alleviates both the primary and secondary clustering problems in linear probing by replacing the constant c with a second hash function $h_2(k)$. Specif-

ically, the double hash function can be written as

$$H(k,i) = (h_1(k) + ih_2(k)) \bmod N, \tag{3}$$

where $h_1(k)$ and $h_2(k)$ are ordinary hash functions which return values in the range $[0, N-1]$. For a given key, these ordinary hash functions yield constants. Using $x_0 = h_1(k)$, $y_0 = h_2(k)$, and $x_i = H(k,i)$ we can rewrite equation (3) as

$$x_i \equiv x_0 + iy_0 (\mathrm{mod} N). \tag{4}$$

Notice that double hashing, as described in equation (4), is the same as the linear hashing, equation (2), except that the constant c has been replaced by the value y_0. Therefore y_0 must always be relatively prime to N in order to guarantee full length probe sequences. The second hash function should be chosen so that all values of $h_2(k)$ are relatively prime to N. The easiest way to assure this is to choose N as a prime number so that any choice of $y_0 = h_2(k)$ in the range $[0, N-1]$ will be relatively prime to N.

The key advantage of double hashing over linear hashing is that y_0 is able to vary with k. Indeed, Knuth [1973] points out that double hashing can provide a good approximation to uniform hashing, and suggests using the double hash function in equation (3) with pairs of primes N and $N-2$, such that

$$h_1(k) = k \bmod N \tag{5}$$

and

$$h_2(k) = k \bmod (N-2). \tag{6}$$

3. CHAOTIC MEASURES AND DYNAMICAL SYSTEMS

The assertion that hash functions and chaotic iterators share some of the same desired properties was put forth by Heileman, Abdallah, Hush, and Baglio [1993], where it was suggested that a chaotic iterator which exhibits sensitive dependence on initial conditions might also perform well as a hash function. The authors introduce the notion that hash functions can be transformed into chaotic iterators in the real domain, allowing some measures from the field of nonlinear dynamics to be applied. This was done by converting the hash functions to iterators in the continuous domain, and then applying the continuous Lyapunov exponent to the resulting iterator [Peitgen et al. 1992]. The results showed that the corresponding double hashing iterator had a positive Lyapunov exponent in the real domain, indicating that this iterator has sensitive dependence on initial conditions. Similar tests for linear probing indicated that it had a zero Lyapunov exponent, or no sensitive dependence on initial conditions.

A general form for a dynamical system is given by the first order recurrence relation

$$x_{n+1} = f(x_n), \quad x_0 = c, \tag{7}$$

where the constant c is the *initial condition*, and $f : \Re \to \Re$. The function f generally must be nonlinear to generate complex behavior. This simple system is called an *iterator*. It is well-known that for some choices of even simple f in equation (7), a system that exhibits extremely complex behavior can be obtained. One such form of behavior is referred to as *chaos*. While a universally accepted

definition of chaos does not exist, it is generally agreed that one characteristic is sensitive dependence on initial conditions, coupled with bounded behavior [Peitgen et al. 1992].

3.1 Lyapunov Exponent

A common technique used to measure sensitive dependence on initial conditions is considered next. Qualitatively, an iterator is said to be sensitive to initial conditions if the orbits that result from two initial conditions, which are arbitrarily close, are distinctly different. The technique most often used to detect this type of behavior involves computing the *Lyapunov exponent* of system (7). Over the real domain, the Lyapunov exponent of system (7) is defined as:

$$\lambda(x_0) = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \ln |f'(x_{k-1})|. \tag{8}$$

The exponent of equation (8) represents the mean exponential rate of divergence or contraction between two nearby orbits. Since $f'(x_{k-1})$ is difficult to calculate for most iterators, the Lyapunov exponent is usually expressed as

$$\lambda(x_0) = \lim_{n\to\infty} \left\{ \lim_{\epsilon\to 0} \left(\frac{1}{n} \sum_{k=1}^{n} \ln \left| \frac{E_k}{\epsilon} \right| \right) \right\}, \tag{9}$$

where $E_k = f(x_k) - f(x_k + \epsilon)$ and $0 < \epsilon \ll 1$. A positive Lyapunov exponent indicates error growth, which means that the iterator being measured is sensitive to initial conditions. A zero or negative Lyapunov exponent indicates either no dependence on initial conditions, or a contractive iterator where small errors are damped with each successive iteration.

3.2 Integer Lyapunov Exponent

We now consider evaluating the Lyapunov exponent over the integer domain Z_N where hashing occurs. Several features of equation (8) must be changed to evaluate the exponent over Z_N. First, the limits can be evaluated only for a finite table size N, and not as $n \to \infty$. Second, the smallest error ϵ which can be resolved for a given x_0 is exactly one. Thus, the best we can do is choose $E_{k-1} = \epsilon = 1$, thereby eliminating the denominator in the sum of equation (9). Taking these differences into account, we can define the integer Lyapunov exponent for a given number of iterations m as:

$$\lambda(x_0, m) = \frac{1}{m} \sum_{k=1}^{m} \ln |E_k| \tag{10}$$

where

$$E_k = |f(x_k) - f(x_{k+1})|. \tag{11}$$

This result can also be reached by noting that in the discrete case, $f'(x_k) = f(x_{k+1}) - f(x_k)$ and directly substituting into equation (8) to get equation (10). This integer Lyapunov exponent can be easily calculated for any integer iterator using equation (10), and is independent of the initial value x_0 for full length probe sequences. The meaning of the exponent has also significantly changed from that

of the real domain Lyapunov exponent, in part because all finite field iterators are necessarily periodic. By definition, the integer Lyapunov exponent produces a positive value for all non-trivial sequences. In fact, the only iterator which will produce a zero Lyapunov exponent is the trivial iterator $x_i = x_{i-1}$. Secondly, for some iterators, the Lyapunov exponent may depend on the number of iterations evaluated, m, as well as the table size N. This is due to the fact that the table size forms an upper bound for the distance between any two values E_k, limiting the value of each term in the summation. Empirically it was found that for most common hash functions, the integer Lyapunov exponent is a function of table size when evaluated with $m = N$ iterations.

The integer Lyapunov exponent, however, does preserve one important characteristic of the real Lyapunov exponent; it serves as a measure of the average distance that very close values will be separated by an average iteration. This is important when the input data distribution is nonuniform, because it is desirable to have similar keys (i.e., keys close in value) distributed in the hash table as widely as possible after only an iteration or two.

3.3 Integer Lyapunov Evaluation

Our first numerical experiments focused on the evaluation of Lyapunov exponents for a variety of iterators and initial hash functions, including variations of linear and quadratic probing, and double hashing. The results of these experiments were, not surprisingly, inconclusive. The relationship between iterator and Lyapunov exponent appeared to be a complex one. All commonly used hash iterators had an integer Lyapunov exponent which depended on table size, yet in many cases the exponent was not directly related to the actual performance of the function for random key values. In some cases a higher integer Lyapunov exponent was associated with a poorer performing hash function. It was concluded that the evaluation of integer Lyapunov exponent alone was not a sufficient measure of hash function performance.

An analytical evaluation of double hashing provides some insight. We can rewrite equations (5) and (6) as

$$H(k,i) = [(k \bmod N) + i(k \bmod (N-2))] \bmod N, \tag{12}$$

where N and $N-2$ are prime numbers. As discussed in Section 2.2, this function produces unique probe sequences for each unique value of $k < N$, and all probe sequences will be of full length. An estimate of the integer Lyapunov exponent can be determined analytically. If one starts with an initial key $k_0 < N-3$, one can analytically perform the summation in equation (10). First, observe that the expressions $k \bmod N$ and $k \bmod (N-2)$ are both equal to the original key k for $k < N-3$. In this case, hash function (12) reduces to

$$H(k,i) = (k + ik) \bmod N. \tag{13}$$

Evaluating the individual terms of the sum for equation (10) yields

$$E_{k_0} = |H(k_0,i) - H(k_0+1,i)| \; = \; |(k_0 + ik_0) \bmod N - (k_0 + 1 + i(k_0+1)) \bmod N|$$

Clearly it is difficult to bound this expression because of the modular reduction operations. However, rough bounds can be established by noting that only when

$H(k_0, i)$ and $H(k_0+1, i)$ are in different epochs modulo N will the difference between the values be greater than $(i + 2)$. Furthermore, for many values of $i > N/2$ the distance E_{k_0} will be substantially less than i because the distance is measured modulo N. Overall, a very rough expected value for the integer error distance E_{k_0} is of order i for the i-th iteration of this hash function:

$$E[\lambda(k_0, m)] < \frac{1}{m} \sum_{i=1}^{m} \ln i = \frac{1}{m} \ln i!$$

where $E[\lambda(k_0, m)]$ denotes the expected value of the Lyapunov exponent for m iterations, starting with key k_0. This expectation is easily verified empirically. For example, for $N = 1823$ the measured Lyapunov exponent is 6.00 versus 6.50 predicted as an upper bound above, measured over the entire problem space. This rough bound is sufficient for the analysis.

An important observation can be drawn from this analysis. Recall that the Lyapunov exponent is a measure of sensitivity to initial conditions. It tells us how quickly data initially close together will be distributed widely in the hash table. From the analysis of the double hashing function, it can be seen that values that start close in the table will differ by no more than about $\sum_i \ln i$ slots in the table after iteration i. This is a significant result if one would like collisions resolved after only a few probe iterations. Consider that even for large tables with high load factors, the average number of probes rarely exceeds 30. Therefore data elements which are clustered initially will likely remain clustered for the first several iterations. This has significant implications for the performance of double hashing of non-uniformly distributed data. A hash function which distributes the data uniformly in the hash table from the very first iteration more closely approximates the *uniform hashing* property.

The problem described above is inherent because of the *linearity modulo* N of the most popular choices for hash functions. Because of the linear relationship modulo N between i and x_i in double hashing (see equation (4)) close key values are separated little during the first probe iterations, unless they happen to cross a modulo N boundary. Correcting this problem requires either the use of nonlinear hash functions modulo N for $h_1(k)$ and $h_2(k)$ or a nonlinear probe function modulo N.

3.4 Entropy

Where the Lyapunov exponent measures distance between subsequent iterations, the information theory measure of entropy measures the information content of the resulting distribution. Both are indicators of overall performance. Entropy, as introduced by Shannon [1948], is a useful measure of the uncertainty of a data distribution.

Definition 1. Suppose X is a random variable which takes on a finite set of values according to a probability distribution $\rho(X)$. Then the entropy of this probability distribution is defined to be:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{14}$$

Entropy can be used to measure the randomness of a finite sequence by measuring the probability of occurrence of each element in the domain, and directly calculating the entropy. The following theorem gives us a theoretical upper limit of the entropy of any finite set:

THEOREM 1. *Suppose X is a random variable having probability distribution $p_1, p_2, \ldots, p_n$, where $p_i > 0, 1 \leq i \leq n$. Then $H(X) \leq \log_2 n$, with equality if and only if $p_i = 1/n, 1 \leq i \leq n$.*

From the above theorem the maximal entropy for a finite domain of size n can be easily calculated as:

$$H(X) = \log_2 n \tag{15}$$

Since the theoretical maximal entropy can be so easily calculated, and that level of entropy occurs only when $p_i = 1/n$, it is easy to determine how close a given distribution is to uniformly distributed data. Entropy measures are frequently applied to cryptographic output as well as random number generators to determine how precisely the output duplicates random data. The same approach can be applied to hash probe functions by measuring the entropy of a large number of probe sequences. We will present the results of this effort in section 5.

4. THE EXPONENTIAL HASH FUNCTION

The Lyapunov measurements above led to the development of a new open address hash function for use on non-uniformly distributed data. As we have discussed, two alternatives exist, either choose nonlinear functions modulo N for $h_1(k)$ and $h_2(k)$, or create a nonlinear modulo N probe function. The problem with the first approach is that the hash functions used in double hashing must be quickly evaluated, yet must also preserve uniform distribution of the hashed data in the table space. It is difficult to create a nonlinear modulo N function that meets all three criterion. A good choice would appear to be $h(k) = k^m \bmod N$ where the exponent m is chosen to be relatively prime to $N - 1$, and N is prime. This function, similar to that used in public key encryption, appears to be a good choice because it yields a permutation of the values $[2 \ldots N - 1]$, it is nonlinear, and it uniformly distributes the data. However, the evaluation of the integer exponent k^m is much more expensive than the simple hash function $h(k) = k \bmod N$, requiring a multiplication and division for each bit of m, [Stinson 1995]. Clearly this would be a poor choice in terms of performance relative to commonly used methods.

What is needed is a hash function that has a large Lyapunov exponent when evaluated over the first few iterations, rather than the entire range of N. This is based on the fact that while double hashing has a nearly ideal Lyapunov exponent when evaluated over the whole table, its worst Lyapunov measure is over the first few iterations. In addition, the hash function should preserve all of the desirable characteristics of double hashing, including fast run time, long probe sequences, and no primary or secondary clustering. The function cannot be a linear function of i, or it would suffer from the same limitations as double hashing. The exponential hash function we propose is

$$H(k, i) = h(k)^i \bmod N, \tag{16}$$

where $h(k)$ is a hash function returning integer values in the interval $[2, N]$. Equivalently, the exponential hash function can be expressed as the iterator

$$x_i = x_0^i \bmod N, \tag{17}$$

where N is prime. This function is similar to the RSA and ElGamal cryptosystems [Stinson 1995], in that a finite field exponent is used to create a nonlinear permutation of values.

This probe sequence has the following characteristics. It can be computed efficiently. The value x_i at the i-th step is simply the previous value x_{i-1} times x_0 modulo N. This requires the same number of mathematical operations as the linear and double hash functions we have described. The probe sequence is nonlinear modulo N. Small perturbations in the initial value x_0 become large differences after only two iterations. The probe sequence depends entirely on the initial hash value x_0, which may lead to primary and secondary clustering. Fortunately this can easily be remedied by adding a second hash function value $h_2(k)$, as will be demonstrated shortly. The probe sequence is not of length N for all values of x_0, since only cases where x_0 is a generator for Z_N^* will generate the full domain.

4.1 Theoretical Performance

Some concepts from number theory will be introduced in order to analyze exponential hashing. First, the concept of a cyclic group must be introduced.

Definition 2. (cyclic group, generator) If a group G contains an element a such that every element of G is of the form a^k for some integer k, then G is a *cyclic group*, and a is called a *generator* of G.

The group Z_p^* consisting of the integers $\{1 \ldots p\}$ and operator $*$, which is normal multiplication modulo p, p prime, forms a cyclic group. In addition we will use the following definitions:

Definition 3. (order) The number of unique elements in a group is called the *order* of the group. The group Z_p^* has order $p - 1$.

Definition 4. (subgroup) A subset H of a group G is a *subgroup* of G, if H is itself a group relative to the binary operation defined on G.

The performance of the exponential hash function depends primarily on LaGrange's Theorem which defines the order of a group of size N, and a corresponding lemma which defines the order of a group in terms of its Euler function.

THEOREM 2. *(LaGrange's Theorem) If G is a group of order N, then the order of every subgroup of G is a divisor of N.*

LEMMA 1. *The number of generators for a cyclic group of order N is $\phi(N)$ where $\phi(N)$ denotes the Euler function — the number of integers less than N which are relatively prime to N.*

Ideally, all of the elements of Z_p^* should be generators of the entire group. This implies that every element can be generated starting with any element. This means that every element leads to a full length probe sequence. Applying the above results to exponential hashing, i is readily apparent that Z_p^* is a group of order $p - 1$ and

that the number of generators for the group is $\phi(p-1)$. Unfortunately p must be prime for Z_p^* to be a cyclic group; which means that $p-1$ must be an even number. Therefore $p-1$ must have as one of its factors the number 2, which leaves at most only $p/2$ elements of Z_p^* as generators of the group.

Next, apply LaGrange's theorem to partially correct this deficiency. Since Z_p^* is a group of order $p-1$, all subgroups H of Z_p^* must have orders that are divisors of $p-1$. However, carefully selecting $p-1$ such that it is the product of 2 and another prime number t will assure that all subgroups of Z_p^* have order of either 2 or t. In fact, if the prime t is chosen carefully so that $p = 2t+1$ is also prime, all elements in the group Z_p^* will either be generators of the entire group, generators of subgroups of order t or generators of subgroups of order 2.

Since the subgroup of order 2 is also cyclic, it has only one generator in Z_p^* It is easy to see that the value $x = (p-1)$ is in fact the only element in Z_p^* which can generate a subgroup of order 2. These results are summarized in the following lemma, which follows directly from LaGrange's theorem with the above choice for p

LEMMA 2. *Given primes p and t, with $p-1 = 2t$, the group $G = Z_p^*$ contains exactly $t-1$ generators for the entire group G, t elements which are generators for subgroups of order t, and one element which generates a subgroup of order 2.*

Therefore, the following conclusions can be derived for the exponential probe function in equation (17) by applying theorem 2. Half $(t-1)$ of the choices for x_0 will be generators for Z_p^*, and will create full length probe sequences. Exactly t of the values for x_0 will generate probe sequences of length t. Only one value, $x_0 = p-1$ will generate a poor probe length of 2. This value can be avoided by choosing the initial hash value $x_0 = (k \bmod (p-2)) + 1$. Different initial values x_0 will generate unique probe sequences in Z_p^*.

The primes t and p can be efficiently generated with $p-1 = 2t$ using probabilistic primality testing. The Prime Number Theorem says that there are approximately $\log(N)$ prime numbers less than N. It follows that the probability of a randomly selected number t being prime is approximately $\log(t)$, and the probability of $p = 2t+1$ being prime is $\log(p)$. Since these are independent events, the probability of both t and p being prime is $\log^2(N)$. Therefore the expectation is that one would have to explore approximately $\log^2(N)$ such randomly selected pairs to find a suitable table size probabilistically. This only needs to be done during initialization of the table.

In summary, this exponential probe function has many desirable characteristics. Except for the less than optimal probe length on 1/2 of the table elements, it has many of the characteristics of double hashing.

4.2 Improvements to the Hash Function

Since the probe sequence generated by exponential hashing depends only on one initial hash value $x_0 = h_1(k)$, it suffers from the same secondary clustering problem as linear hashing. This can easily be corrected by adding a second hash function:

$$H(k,i) = [h_1(k)^i + h_2(k)] \bmod N \tag{18}$$

where ideally $h_1(k) \neq h_2(k)$ for all k. This will assure that no two keys have the same probe sequence, avoiding the primary and secondary clustering problem. Each probe location can be calculated with a single multiplication, addition, and modulo division, which compares favorably with double hashing which requires two additions and a modulo division. Henceforth, we will use equation (18).

4.3 Lyapunov Analysis

An integer Lyapunov analysis of the exponential hash function yields promising results. This hash function can be rewritten as:

$$x_i = (x_0^i + y_0) \bmod N \tag{19}$$

where $x_0 = h_1(k)$ and $y_0 = h_2(k)$. Applying the definition for the integer Lyapunov exponent yields:

$$E_k = |x_i - x_{i-1}| = |(x_0^i \bmod N) - (x_0^{i+1} \bmod N)| \tag{20}$$

It is difficult to establish a tight bound on E_k. Since the exponential iteration is a permutation of all members of the subgroup for which x_0 is a generator, on the average x_0^i and x_0^{i+1} will be evenly distributed in the group. Since all subgroups are subsets of Z_p^*, assuming uniform distribution in $[1, N-1]$ will not be far from the actual distribution. This implies an expected distance of approximately t between subsequent probes in the hash table. This has been verified experimentally with measured average distance only slightly less than t in our experiments.

Comparing this to double hashing, one can clearly see that the exponential hash function is better in the critical first few probes. Recall that the double hash function, equation (12), has an expected inter-probe distance of approximately i for the i-th iteration, while the exponential hash function has an inter-probe distance on average of t for each iteration. This means that for the first m iterations of the double hash function, the expected integer Lyapunov exponent is:

$$E[\lambda(x_0, m)] < \frac{1}{m} \sum_{i=1}^{m} \ln i = \frac{\ln m!}{m} \tag{21}$$

while for the first m iterations of the exponential hash function with table size p and $t \approx p/2$ the expected Lyapunov exponent is

$$E[\lambda(x_0, m)] \approx \frac{mt}{m} \approx \frac{p}{2} \tag{22}$$

Clearly for $m \ll p$ the expression $p/2$ is much larger than $\frac{\ln m!}{m}$. This means the exponential hash function will separate clustered data much more quickly than the double hash probe. In the next section this hypothesis is evaluated experimentally.

5. EXPERIMENTAL RESULTS

In order to test the above hypothesis, we implemented both double hashing and exponential hashing. Table sizes were determined using the double prime criterion, where $N = p = 2t + 1$ is required for the exponential hash function, with p and t both prime. The Miller-Rabin probabilistic primality test was used to determine the next largest prime table size meeting this criterion given a target table size [Stinson 1995]. Based on the earlier analyses for both functions this should produce optimal

Function	*Entropy*
Theoretical	12.313
Random Num Gen	12.234
Double Hash	12.238
Exponential Hash	12.179

Table I. Entropy Comparison of Random Probe Sequences

probe lengths. The complete source code is available with the online version of this paper. Using the C source code, all of the results presented here can easily be verified. In addition, the java applet provides an interactive interface, where the double and exponential hash functions can be compared interactively.

5.1 Entropy Experiments

Entropy measurements were taken experimentally using the double hash, equation (12), and exponential hash, equation (18), functions as well as a uniform random number generator as a control for comparison. In the C source code this experiment is available in the EntropyTest() function in the hash.c file. Variations can be tested by varying the table size, cluster size and position variables in this function. For these measurements, a hash table of appropriate size was created with a counter for each location in the table. In the case of the random number generator, locations were picked at random from the table and the corresponding counter incremented. For the two hash functions, successive probe locations were selected and incremented using the probe function. After a fixed number of probes, the entropy is calculated by dividing the count at each location by the total number of probes to get a probability p_i for each location. The entropy is then:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{23}$$

The initial test included a simple comparison of the probe functions. A total of 5000 probes, each of length 10 from random starting points against a table of size 5087. The results are summarized in Table I. The theoretical maximal entropy for this table size is calculated to be 12.313. All three of the functions tested are close in measurement to the theoretical, confirming that both the double hash function and exponential hash function create nearly uniform data distributions when given a uniformly distributed probe starting point.

In a second entropy test, the hash probe functions were tested from a limited probe starting interval. Specifically, a table again of size 5087 was tested with 5000 probes sequences each of length 10. In this case, however, the starting location for the probe sequences was limited to the first 5% of the domain. This simulates a data cluster located in the first 5% of the data space. The results are presented in Table II. As expected, the hash functions do not perform as well from a limited starting point. For both hash functions, the entropy is significantly lower than the random or theoretical maximal entropy. Further, the entropy for the exponential hash function is significantly higher than the double hash function. This reflects the exponential function's higher Lyapunov exponent for initial probe sequences, and the fact that the exponential function does a better job of uniformly distributing

Function	*Entropy*
Theoretical	12.313
Random Num Gen	12.234
Double Hash	9.996
Exponential Hash	10.728

Table II. Entropy Comparison of Clustered Probe Sequences

probes during initial probe sequences. We will see that this difference is reflected in better exponential hash function performance on clustered data in the next set of experiments.

5.2 Direct Hash Table Measurements

All trial runs involved creating two identical empty hash tables of the same size. Source code for the five experiments presented is available in the hash.c file. Elements chosen at random from the data distributions described below were successively added to the table to achieve a load factor of 95% of the table size. The measure of merit was the average number of probes required per element added. For example, if k elements take a total of m probes, the average probes per element is simply m/k. Samples of this measure were taken every 5% of the table load factor, from 5% to 95%, in order to determine the behavior as load factor increased.

Five experiments are presented here, each compares the double hashing function in equation (12) with exponential hashing, equation (18). The experiments are as follows:

(1) Uniform data distribution over the entire table size — To show that the exponential and double hash functions have statistically equivalent performance for uniformly distributed random initial probes.
(2) Clustered data distribution — To demonstrate improved performance the exponential hash function over the double hash function for tightly clustered initial probes.
(3) Variation of cluster size — To demonstrate sensitivity of the exponential hash function to the size of the initial probe cluster.
(4) Variation of table size with fixed percentage initial probe cluster — To demonstrate sensitivity of the exponential hash function to the table size using a fixed initial probe cluster size.
(5) Variation of initial probe cluster location — To demonstrate the sensitivity of the double hash function to the location of the initial probes.

5.3 Uniform Data Distribution

The control case for this analysis was a series of runs done on a uniform initial probe distribution with a fixed table size. Two identical tables of equal size were created and filled to 95% capacity, one using the double hash function and the other using the exponential hash function. The test was repeated 100 times using a different random number seed for each run to determine if any statistical difference in total number of probes to fill the table could be detected. A table size of 3023 was used for these runs. The source code for this experiment is available in the Test1() function

Measure	*Double*	*Exponential*	*% Diff*
Total Probes	1028281	1010148	1.76%
Avg Probes Per Run	10282	10101	1.76%
Std Deviation	305.0	257.7	-15.51%

Table III. Uniform data comparison

of the hash.c file. One iteration of this experiment can be performed interactively using the interactive java applet by setting the data cluster size to 100% and the table size to 3000. A summary of the results are presented in Table III.

With 100 runs, no statistically significant difference in performance could be detected. The difference in total number of probes between the two functions is only 1.76%, which is significantly less than the 15.51% standard deviation as measured between runs of the same hash function. This means that the exponential and double hash functions are statistically equivalent for randomly chosen uniformly distributed initial keys.

5.4 Clustered Data Distribution

The second experiment involved clustering the initial keys over a sub-interval of the total table size, in an attempt to simulate a dense key cluster. The dense data cluster represents many real world data sets where data is far from evenly distributed. A table size of $p = 3023$ was chosen, and all of the data was chosen at random from a single data cluster of approximately 300 elements from the beginning of the data space. Samples of the average number of probes per data element were taken for every 5% of table size, up to a total table load of 95%. Source code used in this experiment is available in the Test2() function of the hash.c file. This experiment can be performed interactively using the interactive java applet by setting the data cluster size to 10% and the table size to 3000. The results are summarized in figure 1.

Clearly the exponential hash function out-performs the double hash function. For high table, load the double hash function stores data in as little as half the number of probes.

5.5 Variation of Cluster Size

This experiment was similar to the previous one, but in this case the cluster size was varied from 2% to 20% of the overall table size. Again, identical tables were created and populated using both the double hash function and the exponential hash function. Data was taken at random from a cluster of size varying from 2% to 20% of the table size, and the average number of probes per element inserted was sampled for each table to reach 95% capacity. A table size of 2027 elements was used for all experiments. Source code used in this experiment is available in the Test3() function of the hash.c file. This experiment can be performed interactively using the interactive java applet by repeatedly running with cluster sizes between 2% and 20% and a table size of 2000. The results, summarized in figure 2 show that the exponential hash function uses far fewer probes than the double hash function. Furthermore, the relative advantage seems to be larger for more tightly clustered data.

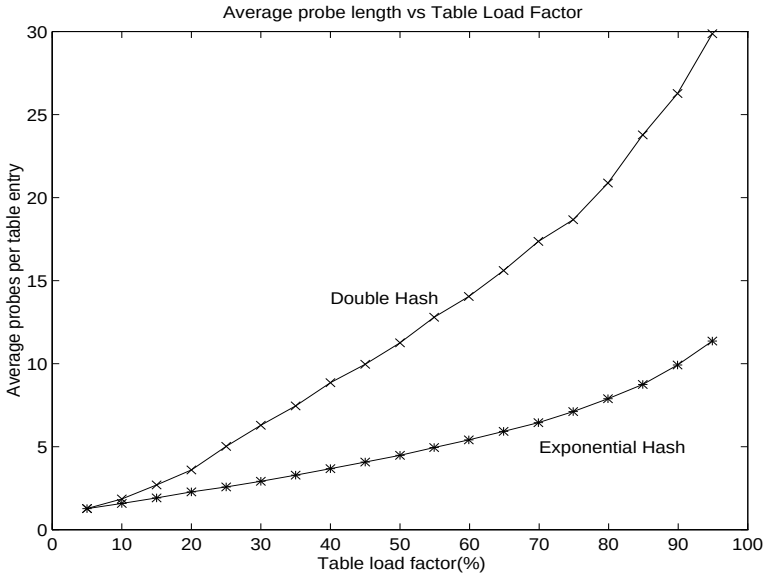

Fig. 1: Average number of probes versus table load factor for the double and exponential hash functions.

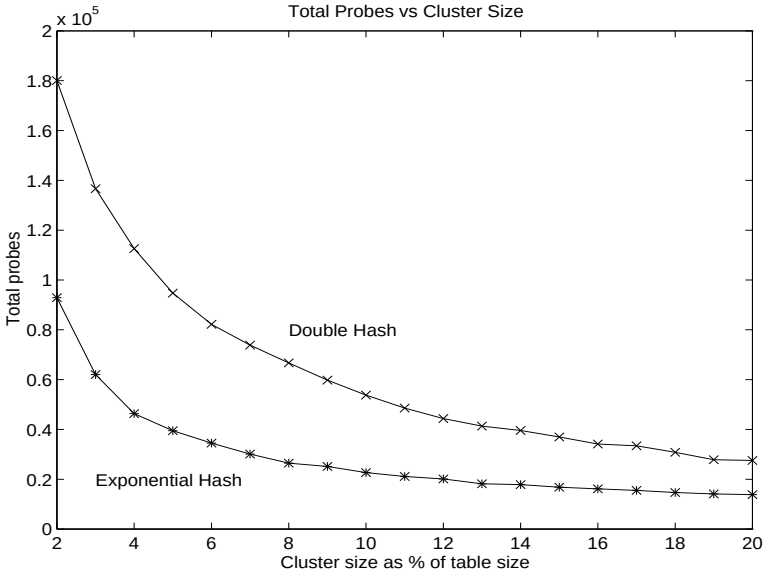

Fig. 2. Total probes versus cluster size at 95% table load.

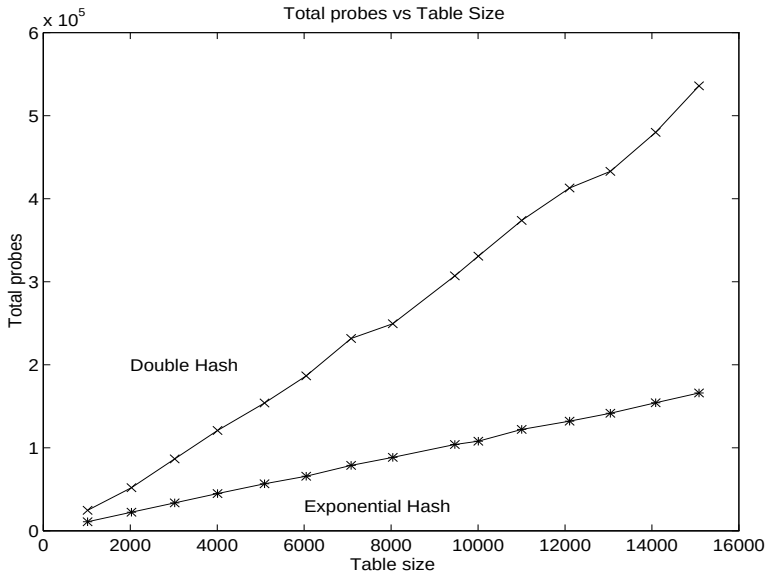

Fig. 3. Total probes versus table size at 95% table load.

5.6 Variation of Table Size

The next experiment varied the table size to see if it had any effect on relative performance of the two functions. Tables were created with from approximately 1000 entries to 15000 entries, and filled to 95% capacity using the double hash and exponential hash function. Results are summarized in figure 3. Source code used in this experiment is available in the Test4() function of the hash.c file. This experiment can be performed interactively using the interactive java applet by adjusting the table size repeatedly to compare performance. The exponential hash function performed better for all table sizes, and table size appeared to have little effect on the relative outcome.

5.7 Variation of Cluster Location

For the final experiment, the location of the cluster in the hash table was varied to determine its effect on hash table performance. A cluster of size equal to 5% of the table size was used for this experiment, and its location was varied from the first 5% to the last 5% of the table in 5% increments. A table size of 2027 was used. Total probes required to fill the table to 95% of capacity was samples for each cluster location. Source code used in this experiment is available in the Test5() function of the hash.c file. This experiment can be performed interactively using the interactive java applet by adjusting the cluster location performance to view the changes in performance as cluster position is moved. The results are summarized in figure 4.

Surprisingly, the double hashing function showed a periodic dependence on cluster location. Data clusters located at the beginning, middle or end of the table led to the largest number of probes while clusters in between these values showed a

much better behavior. The exponential hash function showed much more consistent behavior, with a statistically flat response to cluster location.

We also note that for many data distributions, double hashing performs slightly better than exponential hashing. These instances can be observed in the locations in figure 4 where the number of probes for double hashing falls below that of exponential hashing. From an intuitive point of view, this is not surprising. Recall that average performance for both functions was about the same when uniform data distributions were used. Therefore if we samples all possible data distributions we would expect strengths and weaknesses of each function to average out and approximate the uniform distribution performance. To reconcile this result with the Lyapunov analysis, recall that the Lyapunov analysis made a number of assumptions to simplify the comparison. Chief among these was the assumption that double hashing was approximately linear modulo N, as well as the assumption that exponential hashing produced a uniform distribution of probe locations in N. Clearly both of these assumptions break down for some probe sequences, and some initial data distributions. Nevertheless, we stand by the assertion that a detailed Lyapunov analysis provides valuable information about potential hash table performance for highly clustered data, as well as our claim of more consistent average performance for the exponential hash function presented here.

The reason for the double hash dependence on cluster location has not been fully explored, and is a topic for future research. The shape and periodicity of the curve does, however, seem to indicate some kind of harmonic behavior. Our current theory is that the $ih_2(k)$ term in the double hash function may lead to poor performance if the value $h_2(k)$ is the divisor of a number close to the table size p for small values of i. Preliminary results using other functions for $h_1(k)$ and $h_2(k)$ (i.e. multiplication hashing) indicate that they too have some cluster locations which lead to poor performance, suggesting that this problem may be inherent in double hashing. In any case, these results confirm that the exponential hash function is a better choice for clustered data, since its performance is consistent for all cluster locations.

6. CONCLUSIONS

The results presented here indicate a new relationship between chaos theory and open address hash function performance. Results indicate that the proposed exponential hash function outperforms double hashing for some clustered data distributions, and performs as well for uniform data distributions. These results are confirmed by Lyapunov and entropy analysis as well as actual measured table performance. We believe that there is no significant advantage to using double hashing over exponential hashing. Exponential hashing has the advantage of consistent behavior over all data distributions, while double hashing clearly has poor performance for some clustered distributions. Use of exponential hashing may provide a significant performance advantage where clustered data distributions are common, and its average performance is no worse than double hashing.

A number of avenues for future research are open. It is likely that other measures from nonlinear systems theory can be applied to hash functions, and may also provide additional indicators of hash function performance, possibly leading to further improvements in the exponential open hash function presented here. Furthermore,

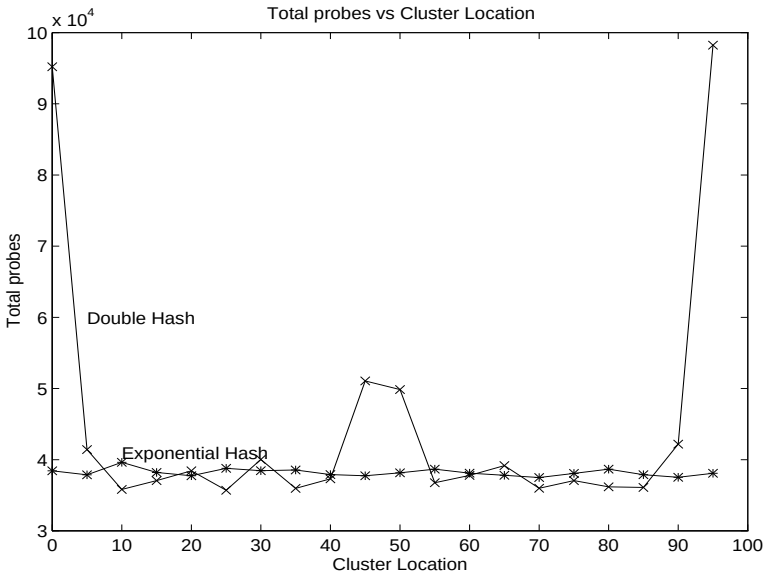

Fig. 4. Total probes versus initial probe cluster location at 95% table load.

it may eventually be possible to apply chaotic measures to other hash function applications such as cryptographic signature verification to detect undesirable hash function characteristics.

REFERENCES

HEILEMAN, G. L. 1996. *Data Structures, Algorithms and Object-oriented Programming.* McGraw-Hill, New York, NY.

HEILEMAN, G. L., ABDALLAH, C., HUSH, D. R., AND BAGLIO, S. 1993. Chaotic probe strategies in open address hashing. In *Proceedings of International Symposium on Nonlinear Theory and its Applications*, pp. 1183–1188.

KNUTH, D. E. 1973. *The Art of Computer Programming*, Volume 3. Addison-Wesley Publishing Co.

PEITGEN, H.-O., JÜRGENS, H., AND SAUPE, D. 1992. *Chaos and Fractals: New Frontiers of Science.* Springer-Verlag, New York.

PETERSON, W. W. 1957. Addressing for random access storage. *IBM Journal of Research and Development 1,* 2, 130–146.

SHANNON, C. E. 1948. A mathematical theory of communication. *Bell Systems Technical Journal 27,* 379–423.

STINSON, D. 1995. *Cryptography Theory and Practice.* CRC Press, Boca Raton, FL.