Nov. 14, 1995

# A Survey of Hash Functions and the Phenomenon of RG

Dong-Keun Shin and Arnold Charles Meltzer

Samsung Electronics
8-2, Karak-Dong, Songpa-Ku
Seoul, Korea

Department of Electrical Engineering and Computer Science
The School of Engineering and Applied Science
The George Washington University
Washington, D. C. 20052

## Abstract

The study surveys several newly developed hash functions along with well-known hash functions such as division, digit analysis, folding, midsquare, multiplicative, radix, random, and Pearson's table indexing. The comparative analysis of the hash coders in a chaining scheme is based on criteria such as distribution, speed, and cost. As a result of this study, a collection of relatively good and data-independent (RGDI) hash functions are now recognized. No noticeable difference has been found in distribution performances of the hash functions in the collection of RGDI hash functions. Based on experimental results in this survey, it is predicted that for most well-defined polynomial time algorithms and intractable problems, there is no distinguishable difference between the performance of one RG (relatively good) solution and that of another. This phenomenon is named the phenomenon of RG.

Among all the RGDI hash functions, the Shin's hash method is not only fast and inexpensive when it is implemented in hardware, but it is also easier and better to use than the well accepted division hash method. Therefore, this paper concludes that the Shin's hash method is a reasonable choice for an effective hash coder in both software and hardware implementation cases.

## 1. Introduction

These days, distributive sorting by a hash function is popularly used in many applications [MAUR1, BABB1, SHIN3]; consequently, there has been a huge demand for an effective hash coder. In some applications, an effective hash coder is essential in increasing the speed of hash-based operations. Another motive for finding a good hash function and the survey of hash functions is that performances of some applications are heavily dependent on distribution performances of a chosen hash function. In searching for a good hash function, one may question about the major criterion for judging a hash function. Therefore, the requirements for a good hash coder need to be clarified first.

The main objectives of a hash function are summarized by Knuth [KNUT1]. Knuth's requirements for a good hash function include the following:

1) computation should be very fast
2) collisions should be minimized.

The first requirement is important in some database applications [BABB1, SHIN3] since the number of keys the hash coder has to transform into hash addresses may be large. The hash address calculation per key is often a main cause of time consumption. Knuth's second requirement for minimizing collisions implies that a good hash function should provide a good distribution performance. Since no hash function can distribute an equal amount of keys into each bucket, it becomes necessary to compare the distribution performance of any new hash function with currently accepted hash functions such as the division method [ULLM1, DATE1, MART1].

According to this survey of hash functions, the distribution performance of some hash functions might show a data dependency problem. In other words, when keys are similar, a data dependent hash function has a larger chance for a collision to occur. Therefore, data independence is a requirement for a good hash function.

When a hash coder is implemented in software, requirements for a hash coder are the same as those for a good hash function, which were discussed above. On the other hand, when a hash coder is implemented in hardware, in addition to the requirements for a good hash function, the requirement of low cost should be satisfied for an acceptable hash coder.

The biggest advantage of a hardware (oriented) hash coder might be speed. This advantage is largely dependent on the kind of hash function chosen. Some hash functions can be accelerated by means of hardware aids; however, others gain relatively little speed even though they cost much more. It is important to determine which hash function fits well into a hardware implementation in terms of both speed and cost, while providing a relatively good, data-independent distribution performance.

The requirements suggested for an effective hash coder implemented in hardware can be summarized as follows:

1. Fast hash address calculation (i.e., a few clock cycles)
2. Relatively good and data-independent distribution performance
3. Low cost in implementation

In this paper, several proposed new hashing functions such as Maurer's shift-fold-loading hash function, Berkovich's Hu-Tucker code hash function, (Additive) Shin's hash function, and Shin's various versions of RC (rotate and combine) are introduced and compared with currently existing hash functions in terms of distribution, speed, and cost.

In Section 2, the experimental environment for this survey of hash functions is explained. In Section 3, the currently existing hash functions and the new hash functions are described. Distribution performances, speeds, and costs of hash functions are discussed in Section 4. Based on the distribution performances of hash functions, the phenomenon of RG is also discussed in this section. Finally, a summary and conclusions are given in Section 5.

## 2. Experimental Environment

The form of hashing considered in this survey is chaining (or open hashing), which provides a potentially unlimited space for each bucket in a hash table. In this hashing scheme, each bucket in the hash table may contain a pointer to a linked list.

In this experiment for a survey of hash functions, three kinds of data sets are used to compare the performance of hash functions. Keys in these three data sets consist of a maximum of 16 ASCII characters; they are left justified and are space character filled. The first data set (RCN) contains 1,024 persons' names, randomly chosen from the phone book, depending on the row, column, and page number, generated by a pseudo-random number generating function. The second data set (GCN) includes 1,024 generally or arbitrarily chosen persons' names with 16 characters. In this data set, there are dozens of groups of people having the same last name. The third data set (RNS) has 1,024 numbers with 16 numeric characters, which are generated by the same function.

Each character in the data sets is internally represented by its corresponding ASCII code. It is assumed that 16 characters in the ASCII code are initially stored in a four-word, or 16-byte, key register. If the ASCII code character string is considered as a number, it may be too large for some hash functions to calculate. Therefore, in this survey, an encoding scheme is used for hash functions such as division, digit analysis, folding, midsquare, multiplicative, radix, and random. On the other hand, hash functions such as Shin's hash function, Maurer's shift-fold-loading hash function, Berkovich's Hu-Tucker code hash function, and Pearson's table indexing hash function do not use encoding schemes. There are many encoding schemes that one can use with a hash function. If a key is encoded into one word, most of the existing hash functions can be directly applied. As Maurer suggests [MAUR2], if keys are longer than one computer word, each word in a key can be folded to the next one consecutively, taking the exclusive-OR. Because this encoding scheme is fast and easily implemented in both hardware and software, it merits attention.

Since this paper also focuses on a fast hardware oriented hash function, calculated hash addresses should be represented with the values of the address bits (8 address bits in this case). The number of buckets in the hash table is 256, 2 to the power of 8. The choice of 256 for the number of buckets in a hash table provides fairness for both hardware- and software-oriented hash functions as indicated by a comparative analysis of their performances.

As the barometer of distribution performance, mean square deviation (MSD) is suggested by A. C. Meltzer. Each hash method is executed on the three data sets to produce mean square deviations, in which the better the distribution, the fewer incidences of collision. Since the number of buckets in the hash table is 256, and because 1024 keys are hashed, a uniform distribution would contain four tuples (the mean in each bucket). The formula of the mean square deviation is:

$$\sum_{i=0}^{x-1} \frac{(Ni-M)^2}{x}$$

$Ni$ : *the number of tuples inserted into bucket* $i$

$x$ : *the number of buckets* (*e.g.*, 256 ($2^8$))

$$M : \text{mean value } (e.g., \frac{\sum_{i=0}^{x-1} Ni}{x} = \frac{1024}{256} = 4)$$

Two speed performances should be determined: one for a software implementation and the other for a hardware implementation. First, when a hash function is implemented in software, execution time in clock cycles can be calculated by hand. The actual instruction-cache case execution time for an instruction sequence of a hash algorithm is derived using the instruction-cache case times listed in the tables of the Motorola Microprocessor User's Manual [MOTO1]. Second, it should be noted that when a hash function is implemented in hardware, the execution time in the clock cycle is calculated for each hash function based on Motorola's HCMOS technology.

The cost of a hardware implemented hash coder is approximately calculated by counting the number of gates used in the coder. Each flip-flop used in either a register or elsewhere is counted for two gates. The gates used for the key register, which is provided to all hash methods, are not included in the number of gates used in the hash coder. If any other device or local memory is used, it is specified in addition to the number of gates by using a postfix mark.

Some hash functions use time-consuming multiplication and division operations. Thus, there is a need for a fast multiplier and divider. A fast modular array multiplier by means of nonadditive multiply modules (NMMs) and bit slice adders, known as Wallace trees, can save time in multiplication compared with an ordinary sequential add-shift multiplier consisting of registers, a shift register, and an adder. A carry lookahead adding divider also substantially increases the speed of a division operation in comparison to the speed of a sequential shift-subtract/add restoring/nonrestoring divider. Hardware organizations of the above multipliers and dividers are explicitly explained in the referenced articles and book [WALL1, CAPP1, STEF1, CAVA1].

According to this survey of hash functions, key-to-address transformation methods are evaluated without weighing other factors such as overflow storage or handling schemes, loading factor, and bucket size owing to the environment of a chaining scheme.

## 3. Description of Current and New Hash Functions

### 3.1 The Division Hash Method

The division hash algorithm [KNUT1, BUCH1, LUM1, MAUR2] simply adds, or exclusive-ORs, the ordinal number of words in a key and takes the remainder, and divides the sum (the combination or the encoded key, Key) by bucket size number **b**. The resulting remainder (h(Key)) could represent any bucket number 0 through **b**-1. Buchholz and Maurer suggest that the divisor should be the largest prime number smaller than **b** [BUCH1, MAUR1].

### 3.2 The Digit Analysis Hash Method

The digit analysis hash method [MAUR2, LUM1] differs from all others in that it deals

only with a static file where all the keys in an input file are known beforehand. Therefore, using either mean square deviation or standard deviation, the skewed distribution of each digit or bit position can be analyzed. The digits that have the most skewed distributions (larger deviations) are deleted to make the number of remaining digits small enough to produce an address in the range of the hash table. This statistical analysis does not guarantee uniform distribution; however, it does provide a better chance of producing uniform spread.

### 3.3 The Folding Hash Method

In the folding hash method [MAUR2, LUM1], the key is partitioned into several parts; e.g., 3 partitions in the key are folded inward like folding paper. Subsequently, the bits or digits falling into the same position are exclusive-ORed (or added). The **k** bits in the resulting partition are then used to represent a hash address for the hash table that has two to the power of **k** (2**k) buckets. This folding method is specifically called 'fold-boundary' or 'folding at the boundaries.'

In another folding method, all but the first partitions are shifted so that the least significant bit of each partition lines up with the corresponding bit of the first partition. Then, these partitions are folded. This method is often referred to as 'fold-shifting' or 'shift-folding.' Several versions of RC (rotate and combine) are developed and discussed in this paper.

### 3.4 The Midsquare Hash Method

In the midsquare hash method [MAUR2, LUM2], the key is multiplied by itself or by some constant, then an appropriate number of bits are extracted from the middle of the square to produce a hash address. If **k** bits are extracted, then the range of hash values is from zero to 2**k - 1. The number of buckets in the hash table must also be two to the power of **k** when this type of bit extraction scheme is used. The idea here is to use the middle bits of the square, which might be affected by all of the characters, or the whole bytes in the key in producing a hash address.

### 3.5 The Multiplicative Hash Method

A real number **C** between 0 and 1 is chosen in the multiplicative hash method [MAUR1, KNOT1]. The hash function is defined as truncate (**m** * fraction(**c** * Key)), where fraction(x) is the fractional part of the real number **x** (i.e., fraction(x) = **x** - truncate(x)). In other words, the key is multiplied by a real number (**c**) between 0 and 1. The fractional part of the product is used to provide a random number between 0 and 1, dependent on every bit of the key, and is multiplied by **m** to give an index between 0 and **m**-1. If the word size of the computer is 32 (2**5) bits, **c** should be selected so that 2**(2**5)*c is an integer relatively prime to 2**(2**5); **c** should not be too close to either 0 or 1. Also if **r** is the number of possible character codes, one should avoid values **c** such that fraction ((**r**p)*c) is too close to 0 or 1 for some small value of **p** and values **c** of the form **i** / (**r** - 1) or **i** / (**r**2 - 1). Values of **c** that yield good theoretical properties are 0.6180339887, which equals (sqrt(5) - 1) / 2, or 0.3819660113, which equals 1 - (sqrt(5) - 1) / 2.

### 3.6 The Radix Hash Method

In the radix hash method [MAUR2, LUM2], a number representing the key is considered

as the number in a selected base, e.g., base 11 rather than its real base. In the radix hash method, the resulting number is converted to base 10 for a decimal address. For example, the key 7,286 in base 10 is considered as 7,286 in base 11; therefore, 7,286 in base 11 becomes 9,653 in base 10, as is shown in the equation below:

$$7*11^3 + 2*11^2 + 8*11^1 + 6 = 9653 \ (base\ 10)$$

Furthermore, the resulting number 9,653 can be divided by the number of buckets in the table. The remainder is then used as a hash address just as in the division method. This combination of two methods, the radix transformation and division methods, is derived from Lin's work [LUM1].

### 3.7 The Random Hash Method

This random hash method [MAUR2] requires a statistically approved pseudo-random number generating function. After the key is encoded, the encoded word is sent to the random number generating function as the seed. Then the random hash method applies division, or some other method, to the generated random number to produce a hash address. The distribution performance of this hash function is thus dependent on the chosen pseudo-random number generating function.

### 3.8 The Pearson's Table Indexing Hash Method

Recently, Pearson introduced a new hash method [PEAR1] for personal computers that lacks hardware multiplication and division functions. The major operations used in this hash method are exclusive-OR and indexed memory read and write. As shown in Figure 1, an auxiliary table(T) is used to contain 256 integers ranging from 0 to 255. Pearson's hash function receives a string of characters in ASCII code. Each character (C(i)) is represented by one byte, which is used as an index in the range 0-255.

As shown in Figure 2, each character of a key is exclusive-ORed with an indexed memory read (H(i-1)) in table H. The resulting byte is used to index the table T, and the indexed value in T is then stored to H(i) for the next iteration step. After the looping process is finished, the last indexed value (H(n)) from table T becomes the hash address for the buckets ranging 0 through 255.

### 3.9 Maurer's Shift-fold-loading Hash Method

Maurer's shift-fold-loading hash method is a hardware-oriented system. The three primary operations in this hash method are shift (or rotate) right, exclusive-OR, and load into a register. All three are relatively fast operations. A key register contains bit information of a whole key. It is the same size register as the key register for fast shift operations, and a number of exclusive-OR gates (one gate for each bit in the key) are required in the hash coder.

Initially an input key exists in both shift and key registers. The shift register will rotate the bit contents one bit to the right; therefore, the rightmost bit will be stored in the leftmost bit in the shift register. Then every pair of bits that are in the same position as the key and the shift registers are exclusive-ORed together. Finally, the resulting bits are loaded into both the shift and the key registers. The algorithm is shown in Figure 3.

As specified in the algorithm in the second rotation, all of the bits in the shift register are rotated three bits right, and exclusive-ORing and loading is followed by the same method as described above. Then the algorithm rotates seven bits right, while performing the same exclusive-ORing and loading once again. It then rotates another 15 bits right and repeats the process. After that, the same process for 31, 63, and 127 bits is duplicated in order. If there are N bits in a key, log N numbers of shift, exclusive-OR and load operations are required, since

$Ki = 2^i - 1 < N \ (i \geq 1)$. *Thus*, $1 \leq i < \log (N+1)$.

### 3.10 Berkovich's Hu-Tucker Code Hash Method

In Berkovich's Hu-Tucker code hash method, the Hu-Tucker variable length code [KNUT1], as shown in Figure 4, is used. Converting each character in a key to its corresponding Hu-Tucker code and storing the binary string of the code for each character, the Hu-Tucker code string for the whole key is accumulatively created, character by character. For example, the Hu-Tucker coded value of the key "ABC" is "0010001100001101". In the conversion process, the string size of a code for each character must be added to provide the total number of bits in the final string of the code. This resulting string of bits is partitioned into substrings which are the same length as a hash address. The last substring might be shorter, but it is filled with zeros. These substrings are folded one by one by taking exclusive-OR. The bits in the resulting string represent a hash address.

For nonalphabet characters, similar patterns based on the idea of Hu-Tucker code can be used. If the sizes of input strings are relatively short while the size of a hash table is large, an extended Hu-Tucker code table can be created based on the idea of Hu-Tucker code.

The principle behind this hash method may be described as the variable length and irregular pattern of the Hu-Tucker code, for each character helps randomize the bit values of a hash address when the fixed length substrings are folded.

### 3.11 Shin's Hash Method

The Shin's (mapping) hash function converts or maps in parallel the internal representation, e.g., ASCII or EBCDIC code, of each character in a key to an arbitrarily chosen prime number (or a randomly chosen number). It then folds these numbers using arrays of exclusive-OR gates to produce a number in binary form and, once again, in a parallel manner. Then the function extracts k bits from the binary number in order to produce a hash address for the hash table that has two to the power of k buckets. Shin's (mapping) hash coder requires hardware components, for example, sixteen 64*16 bits ROMs (one ROM for each character) and eight exclusive-OR or EX-OR modules (120 exclusive-OR gates in total) as shown in Figures 5 and 6. In each ROM, 64 (2**6) arbitrarily selected prime numbers are stored. (Each ROM may contain 128, 32, or 16 numbers if a user chooses 128*16, 32*16, or 16*16 ROM respectively.) The selected numbers should be sufficiently large to be greater than the number of buckets in a hash table. A set of all 16 ROMs is included in the hardware Shin's (mapping) hash coder and the contents of all 16 ROMs are different. In this hash coder, only the least significant six bits of an ASCII character are used as an input address to the corresponding ROM.

As shown in Figure 5, the first bits of the 16 prime numbers are exclusive-ORed together to generate the first bit of a hash address. Simultaneously, the second bits of the 16 prime numbers are exclusive-ORed together producing the second bit of the resulting hash address. All other bits of the hash address are also constructed at the same time. The circuit of EX-OR Module for each bit shown in Figure 5 is represented in Figure 6. The concurrent processing of looking up random numbers and of bit calculations for a hash address increases the speed of hash address computation. The major operations in this hash method are indexed memory read and exclusive-OR. These operations also are time-saving operations. The conversion of an ASCII character to a prime number is a useful aid when randomizing the value of bits. It is, however, necessary to be cautious about designating the least significant bit of every prime number '1', since prime numbers are odd numbers. Consequently, the least significant bit of every prime number should be excluded in forming a hash address. However, one may subtract one from every prime number of evenly numbered slots in each ROM to avoid the problem; accordingly, half of the prime numbers should be decremented by one. If one decides to use a random number generator to fill up the table or the ROMs with random numbers, it would be a good idea to use multiple random number generators to avoid possible dependence on the performance of a chosen random number generator.

By using available instructions, this hash method also can be implemented in software. The algorithm of this hash method in a Pascal-like notation is shown in Figure 7. The programming language of Pascal provides an ORD function which converts a character to a corresponding ASCII integer number. In the algorithm, only the six least significant bits of the ASCII numbers are used as indices to the table containing 64 (2**6) prime numbers.

If the Exclusive-OR (EX-OR) function is to be explained in high level terms, it receives two integer numbers to be exclusive-ORed, which then converts them to two strings of 1's and 0's, and takes the exclusive-OR on the bits of the same position in the two strings. The Shin's hash function then converts the resulting binary string back to integer output to be sent to the calling program. In the hardware implementation of this Shin's hash method, the exclusive-OR operation is more valuable than the addition operation due to the fact that the exclusive-OR operation does not generate a carry-out bit. Should anyone implement this hash function in a high level language while disregarding speed, additive Shin's hash function, which employs the addition operation, could be used instead. The next section explains the Additive Shin's hash function, illustrating it's simplified algorithm.

If the least significant **k** bits from the sum or the combination can be extracted in order to produce a hash address for the table of 2**k buckets, the time-consuming MOD operation that provides a remainder after a division is rather unnecessary. This alternative method is acceptable, since the sum or the combination in the variable Temp is already adequately randomized.

The following statement in the algorithm, "Temp := EX_OR (Prime_Table (Index), Temp);" performs the exact same function that the hardware implemented Shin's hash method does. If '+' is understood to represent exclusive-OR operation on two input bits, and X1 is the first bit of the first prime number, then X2 is the first bit of the second prime number, and so on. As a result, Xi is the first bit of the i-th prime number. The assertion can be expressed with the following equation:

(((X1+X2)+(X3+X4))+((X5+X6)+(X7+X8)))+
(((X9+X10)+(X11+X12))+((X13+X14)+(15+X16)))
is equal to (=)
((((((((((((X1+X2)+X3)+X4)+X5)+X6)+X7)+X8)+
X9)+X10)+X11)+X12)+X13)+X14)+X15)+X16

The left-hand side of the equation represents how the parallel exclusive-ORs on the first bits are taken from the 16 prime numbers. The right-hand side of the equation represents how the serial exclusive-ORs on the first bits are taken from the 16 prime numbers. By using either the associative law of exclusive-OR (such that (X+Y)+Z = X+(Y+Z) = X+Y+Z) or a lengthy truth table, one can prove that both sides of the above equation are equal. Shin proved it using the associative law of exclusive-OR [SHIN1]. Thus, it is obvious that the parallel and serial processing of the Shin's hash method are equivalent. Using the same proving technique, the parallel exclusive-ORs and serial exclusive-ORs can be proved to be equal. People can also logically infer that the same principle works when the number of input bits is not exactly two to the power of a constant.

Considering the limited bandwidth in transferring keys, character by character (or word by word) serial processing for hashing a key is also feasible using an iterative hardware component.

Although input keys are not character strings but others, such as integer numbers, each key might be either converted to a character string or considered as a character string that is composed of several bytes. Therefore, this hash coder can be employed with little or no modification to hash keys in any key sets regardless of the input format of their keys.

### 3.12 Additive Shin's Hash Method

The algorithm of additive Shin's hash method employs the addition operation instead of the exclusive-OR operation that the Shin's hash method employs. When one implements the additive Shin's hash function in a high level language, the following statement can be used: "Temp := Prime_Table [Char_No, Index] + Temp;" in place of the statement: "Temp := EX_OR (Prime_Table [Char_No, Index], Temp);" in the algorithm shown in Figure 5. After the for-loop in the algorithm shown in Figure 8, the sum in the variable Temp becomes an adequately randomized (hashed-up) value. The sum will be divided by the number of buckets, and the remainder will be used as a hash address. This hash method gives as good a distribution performance as the Shin's hash method, as shown in this survey. Since both Shin's hash method and additive Shin's hash method require a multitude of prime numbers, an algorithm for finding prime numbers within a specified range is provided by Dong-Keun Shin and presented in Figure 9.

### 3.13 Shin's RC Hash Method

As has been shown by several researchers [KNUT1, MAUR2, KNOT1, LUM1], the fold-shifting hash method is the fastest and most easily implemented method in hardware. In hardware implementation of the fold-shifting hash method, the original encoded keyword can be shifted, not by a shift register, but by wires that are shifted in their connection to exclusive-OR gates.

When there is an encoded one word key or a key with several partitioned words, there may be many ways to fold using the exclusive-OR operation. The two major questions on the fold-shifting method are as follows:

1) How many partitions have to be made on a key?
   (Or how many folding processes are needed?)
2) How many bits should be shifted or rotated for each partition?

In answering the above questions, it is necessary to consider how many shifted keywords are needed in folding in order to randomize the bits in the resulting word. Each byte in an encoded keyword may have a similar pattern. However, the pattern in each byte should be eliminated in the folding process. Hence, the scope of randomization is narrowed down to a byte. If the number of bits rotated is one, then eight rotated keywords might be sufficient to randomize every bit in a byte, since eight, the number of keywords, times one, the number of bits rotated, is the number of bits in a byte. This fold-shifting process may be represented by Rot(0,1,2,3,4,5,6,7).

If the number of bits rotated is two, then the four rotated keywords may be enough to randomize every bit in a byte, since the number of bits to be rotated, two, times the number of rotated keywords, four, is the number of the bits in a byte. For example, Rot(0,2,4,6) is equivalent to any combination of 0,2,4, and 6, e.g., Rot(2,4,6,0), Rot(4,6,0,2), etc. Rot(0,2,4,6) also is symmetric to Rot(1,3,5,7), because their resulting bits are only ordered differently. It becomes evident that the number of rotated keywords required is the upper boundary of the number that results from the number of bits in a byte, eight, divided by (/) the number of bits rotated. For hardware implementation, it would be preferable if the number of rotated keywords is 2**r (r=1, 2, or 3), due to the fact that each exclusive-OR gate has two inputs.

When the number of bits rotated is three, Rot(0,3,6,1) would be considered. If the number of bits rotated is four, Rot(0,4,1,5) can be used instead of Rot(0,4,0,4) or Rot(0,4). If five bits are rotated, Rot(0,5,2,7) can be used. When six bits are rotated, Rot(0,6,4,2) would be considered; however, it is symmetrical to Rot(0,2,4,6); therefore, Rot(0,6,4,2) would not be selected. If seven bits are rotated, Rot(0,7,6,5) can be used.

These fold-shifting hash methods may require that the number of rotated keywords to be four (2**r, r=2) because two is too little and eight is too many. Interestingly, there are four bytes in an encoded keyword, and the number of rotated keywords are four. Therefore, it can be deduced that at least some portion of each byte should affect the other three bytes in the keyword. Accordingly, eight bits should be rotated right in the second keyword, 16 bits should be rotated right in the third keyword, and 24 bits should be rotated right for the fourth keyword. Thus, Rot(0,2,4,6) becomes RC (0,10,20,30) (i.e., RC(0+8*0, 2+8*1, 4+8*2, 6+8*3)). By the same process, Rot(0,3,6,1) becomes RC(0,11,22,25) (i.e., RC(0+8*0, 3+8*1, 6+8*2, 1+8*3)), Rot(0,4,1,5) becomes RC(0,10,17,29), Rot(0,5,2,7) becomes RC(0,13,18,31), and Rot(0,7,6,5) becomes RC(0,15,22,29). Likewise, Rot(0,4) which combines two rotated keywords may become RC(0, 4+8*1), RC(0, 4+8*2), or RC(0, 4+8*3). Rot(0,1,2,3,4,5,6,7) which combines eight rotated keywords may become RC(0;1,2,3,4,5,6,7), RC(0+8*0, 1+8*1, 2+8*2, 3+8*3, 4+8*0, 5+8*1, 6+8*2, 7+8*3), or similar rotation patterned RC hash functions. One may create more RC hash functions using the rotate and combine technique. In this paper, the RC2, the RC4, and the RC8 (rotate and combine two keywords, four

keywords, and eight keywords respectively) are specifically described.

## 4. Discussion With An Analysis of Distribution, Speed, and Cost

Table 1 shows each hash function's performance in terms of distribution, in terms of speed when implemented either in software (SW) or in hardware (HW), and in terms of the cost of the hardware implementation of the hash function. As a measurement of distribution, mean square deviation (MSD) is provided whenever a hash function is applied to the three different data sets: randomly chosen names (RCN), generally chosen names (GCN), and randomly chosen numeric strings (RNS). The number of clock cycles (clocks) is used in measuring the speeds of the hash coders. The cost of building a hardware hash coder is roughly represented according to the number of gates needed. Experiments on hash functions to show their distribution performances were conducted by Shin, and the speeds and costs of the software and hardware hash coders were measured by Shin [SHIN1] as well.

Distribution performances of the Shin's hash method have been developed in cases when each ROM contains prime numbers and when each ROM contains random numbers. As shown in the Tables 2A and 2B, mean square deviations hover around four, as do those of other relatively good hash methods. Since there is no distinguishable difference between using prime numbers and random numbers for each ROM, there is no clear reason to insist on solely prime numbers. The results do not provide any clue regarding data dependency since the Shin's hash function distributes numeric string keys as well as other keys. Different groups of eight bits, e.g., 1-8, 2-9, 3-10, 4-11, 5-12, 6-13 bits, are extracted to compose a hash address (The 1-8 means bits 1 through 8 are selected.). In summary, there is no noticeable difference between the distribution performances of the various groups.

By virtue of byte-by-byte parallel processing, with separate ROM and exclusive-OR module, the Shin's hash method can produce a hash address within three clock cycles (The calculation time is measured after a key string is loaded into the key register.). The Shin's hash method requires as many clock cycles as other hash methods do when loading a key string into the key register. Two clock cycles are required in order for the memory read to retrieve a random number from the corresponding ROM. One clock cycle is needed for the calculation process of hash address bits through the four levels of exclusive-OR gates. The maximum gate delay is nine nanoseconds and the clock frequency is set to 20 MHz (50 nanoseconds per a clock pulse width); thus, the address bit signal can pass through the four gate levels ($4*9 = 36 < 50$ nsec) within a clock cycle.

Based on the stored contents (selected prime numbers) of the ROMs, each Shin's hash coder calculates a hash address in its unique way. The hash addresses, generated by different Shin's hash coders, are independent of each other, but the address calculation time for each hash coder is always the same. It is this characteristic of functional difference that constitutes the asset of the Shin's hash function. This property also is valuable in an application environment which uses parallel hashings or a rehashing scheme. It is noteworthy that the additive Shin's hash method shows competitive distribution performances (MSDs of 4.40, 3.39, 3.58) when it is tested. This result supports the claim that addition and exclusive-ORing produce the same effect in randomizing the bit values.

The selected RC hash methods to be examined are RC4 hash functions such as

RC(0,10,20,30), RC(0,11,22,25), RC(0,10,17,29), RC(0,13,18,31), and RC(0,10,17,29). There are several reasons for examining these hash functions. One of the reasons for examining these hash functions is to verify that the phenomenon of RG is observable in distribution performances of the hash functions. By testing the RC4 hash functions and comparing them with complex hash functions, it is possible to be assured of the fact that simple hash address calculations can distribute keys as good as complex calculations (e.g., division and multiplication) can. Some of RC4 hash functions are good to be RG (relatively good) hash functions, but some of them cannot be an RG solution. People will realize that the phenomenon of RG is true, because there is no difficulty in distinguishing RG solutions from poor solutions. Moreover, they can observe that because of the phenomenon of RG, there is no noticeable difference in the distribution performances of the RGDI hash functions except small variations (which can be neglected) dependent on different input data sets.

The distribution performances of Shin's RC hash method, in particular, RC(0,10,20,30) and RC(0,11,22,25) are as good as those of other acceptable hash methods. But other selected RC methods, such as RC(0,12,17,29), RC(0,13,18,31), and RC(0,15,22,29), show a data dependency problem, where the distribution performance on the RNS data set is not compatible with the distribution performance on the RCN and GCN data sets, as is demonstrated in Table 3. Therefore, when using this hash method, careful selection of the number of partitions and the number of rotated bits is required. The performance of a hardware hash coder is dependent on the randomness of each bit value in produced hash addresses. For example, if a single bit is stuck at either '0' or '1' for all the produced hash addresses, half of the buckets in the hash table will be empty. Hence, for a good hardware hash coder, the value in each bit position of produced hash addresses should not be stuck at either '0' or '1'. The RC2 (rotate and combine two keywords) is inexpensive but it may distribute keys poorly compared to other RC hash functions. On the contrary, the RC8 (rotate and combine 8 keywords) is more complex and expensive than other RC hash coders, but it may have more reliable distribution performance than other RC hash coders have.

Hardware oriented hash methods such as fold-shifting and Shin's RC have a problem when they are implemented in a high-level language, because high-level languages usually do not provide operations such as shift and rotate word. However, the Shin's RC is recommended for a hash coder when an inexpensive and very fast hardware hash coder is required.

The distribution performance of the division hash method [BUCH1, MAUR2, LUM1] varies depending on the chosen divisor which is close to the number of buckets, as is shown in Table 4. If an inappropriate divisor is chosen, a data dependency problem may occur. In this experiment, the divisors that are greater than the number of buckets in a hash table (i.e., 256) are tested, also. Therefore, if a produced value for a hash address is greater than the number of buckets, the value is folded inward at the boundary of the hash table to find a substitute. Some odd numbers employed as a divisor of a division hash method are carefully examined. The divisor 257 is a nonprime number with prime factors less than 20. The division hash function which uses the divisor 257 still shows very poor distributions (MSDs of 5.67, 11.95, and 122.99). As Maurer and Buchholz suggested [MAUR1, BUCH1], using the largest prime number, (i.e., 241) that is also smaller than the number of buckets, as the divisor, yields better results (MSDs of 5.51, 5.35, 4.48).

We disapprove of the additive division hash method which sums the integers (ASCII

values) of each character and divides the result by the number of buckets(B), taking the remainder, which is an integer from 0 from B-1. The experimental results of distribution performances of this hash method are MSDs of 3.97, 3.91, and 86.76. In this case, the addressing range of the division hash method is very limited. Accordingly, this division hash method results in a poor distribution when the number of buckets in a hash table is larger than the range. As a result, hash functions with the table-size dependency problem are not recommended.

Several other researchers [BUCH1, LUM1, RAMA1] conducted experiments on typical key sets in order to discover the ideal hash method. Their overall conclusions verify that the simple method of division seems to be the best key to address transformation technique when computational time is not critical. Nevertheless, in this survey of hash methods, the division method is not highly recommended because either the Shin's or the additive Shin's method can be used instead, depending on the application environment. In the application, where fast hash address calculation is not required, the additive Shin's method is superior to the division method. When using the additive Shin's method, one will find that selecting a correct divisor is not a worry; one need only to divide the sum or combination by the number of buckets in order to arrive at a remainder for a hash address. On the other hand, when the speed in address calculation is imperative and the number of buckets can be 2**k, a hardware hash coder is needed. In this case, the Shin's (mapping and RC) hash coders, which are faster and cheaper than the division hash coder, are highly recommended.

Pearson's table indexing hash method appears to be erratic, owing to its poor distribution performance. The fold-boundary and the midsquare both show data dependency problems as in Tables 5 and 6 respectively. In particular, the multiplicative, the radix, and the random hash functions show signs that they may perform poorly for specific data sets. Distribution performance of the digit analysis hash method is measured by using two types of encoded keys: 2 bytes and 4 bytes as shown in Table 1. The findings indicate that this hash method may be data dependent. Both Maurer (see Table 7 for more information) and Berkovich present new hash methods that have proved to be proficient in distribution performance. Their methods, however, have not been highly recommended for the effective hash coder due to their relatively slow hash address calculation speeds.

Hash functions such as midsquare, multiplicative, radix, and random use complex mathematical operations, e.g., multiplication and division. Their speeds of hash address calculation can be increased by fast multipliers and/or dividers. These fast multipliers and dividers, however, are quite expensive. Since there are speed versus cost trade-offs, any judgement regarding adaption must be made thoughtfully. For that reason, the gates of these options are also reflected in the costs of a hash coder in order to help a computer designer make the best decision.

As a result of the survey, a collection of relatively good and data-independent hash functions has been recognized. The Shin's and the additive Shin's hash functions, the shift-fold-loading hash function, the Hu-Tucker code hash function, Selected Shin's RC hash functions (e.g., RC8, RC(0,10,20,30), and RC(0,11,22,25)), and the division hash function are called RGDI hash functions. It is, though, assumed that a RGDI hash function which uses the encoding scheme cannot be a perfect data-independent hash function. For example, the division method is not perfectly data-independent because it uses an encoding scheme. Based on the

chosen encoding scheme, if a data set includes only the keys which are composed of two kinds of characters such as '0' and '1' (e.g.,"1001110100101001", "1100100010100110", etc.) after a encoding process, all bit values in an encoded word will be fixed (i.e., stuck) except four least significant bits for each byte in the four-bytes encoded word. The performance of the division hash method will be very poor in this case. Therefore, the hash function that uses an encoding scheme is not recommended when a data set includes keys which are represented with a very limited number of characters (i.e., less than 10 characters).

The data dependency in the distribution performances of hash functions was first observed and stated by Shin. This survey also verifies that there is no distinguishable difference between distribution performances of RGDI hash functions. This phenomenon was first observed and mentioned by Shin, and again stated in the manuscript which was submitted to the Communications of the ACM in 1991 [SHIN2]. A collection of RG (relatively good) solutions can exist for a problem and there is no distinguishable difference between the algorithmic performance of one RG solution and that of another. A phenomenon of RG is predicted in solutions including both polynomial time algorithms and nondeterministic polynomial functions. Although there are limited variations in performances of RG solutions, they are not caused by algorithmic performances of chosen RG solutions but mainly caused by given input data sets. That is, there may be a collection of RG solutions for some intractable problem and they perform almost equally. One may prove this assertion with the results of rigorous experiments, focusing on some well-known intractable problems.

Based on the phenomenon of RG, people can estimate the performance of an algorithm and the upper bound of the time complexity of the algorithm, considering those of other accepted RG methods. For example, RC8 hash function may have more reliable distribution performance than other RC hash coders have, due to the phenomenon of RG, there will be no noticeable difference between the distribution performance of RC8 and that of any chosen RGDI hash function. The phenomenon of RG will be more obvious if well-chosen criteria for the survey are very few. For example, in this survey, mean square deviation is the sole criterion for this survey of distribution performances of hash functions and it is well chosen; therefore, the phenomenon of RG is quite observable. Once RG solutions are collected, one may choose one not based on the algorithmic performance but based on the merits (e.g., time complexity, characteristic of parallel processing, less I/O overhead, simplicity of the algorithm, feasibility in software or hardware implementation, and efficiency of the algorithm in a given computing environment) of one over those of others.

Due to the RG phenomenon shown in this survey of hash functions, one does not need to ask either if there is a hash function which can distribute keys equally in the buckets or if there is a hash function which obviously outperforms the RGDI hash functions in distributing keys. Our answer to the questions is certainly negative because the RG phenomenon sets a limit on the distribution performance.

## 5. Summary and Conclusions

### 5.1 Summary

If huge amounts of data pass through a hash coder, the hash address calculation should be very fast. In order to speed up the hash address computation, efforts should be concentrated

on designing a new hash function that will avoid time-consuming serial and/or iterative computations while taking advantage of parallel processing, by means of hardware, for converting a key into a hash address. Moreover, the new hash algorithm should distribute random keys into buckets as uniformly as possible. It can be plainly seen that the ideal hash function design for this application is data-independent and calculates a hash address within a few machine cycles with relatively good distribution.

Most of the well-known hash functions, and several new ones, including the Shin's, the additive Shin's, Maurer's shift-fold-loading, Berkovich's Hu-Tucker code, and various versions of RC, were surveyed in this paper. Each hash function has been simulated and applied to two different name data sets (RCN and GCN) and one numeric string data set (RNS) to produce distribution performances measured in terms of mean square deviations. The speed of calculating a hash address is measured in terms of clock cycles for each hash function in both the hardware and software implementation cases. The cost of the hardware implemented hash coder may be calculated and stated in terms of the number of gates used.

As the results illustrated in the tables above indicate, some of the well-known hash functions, such as the midsquare and the fold-boundary, show data dependency problems. Other hash functions, like the multiplicative, the radix, and the random, show signs that they may perform poorly for specific data sets. New shift-fold-loading and Hu-Tucker code hash methods have good distribution performances, but they are not fast in hash address calculation. Not every new RC hash method is reliable in terms of distribution performance; nonetheless, these methods are extremely fast for they require only a single clock cycle after the key is encoded and loaded into the key register. Moreover, the RC hash coder is inexpensive. This survey also shows that there is no distinguishable difference between the distribution performances of relatively good, data-independent hash functions, such as the Shin's, the additive Shin's, the shift-fold-loading, the Hu-Tucker code, RC(0,10,20,30) and RC(0,11,22,25), and the division methods. In this survey of hash methods, the difference of distribution performances between the RGDI hash functions is hardly perceptible. By comparing the distribution performance of a hash function with the performance of other acceptable hash functions, one can easily discern whether or not a hash function can equal RGDI hash functions in distribution performance. This study collects the hash functions together according to distribution performance. As a result of the survey of hash functions, a collection of the relatively good and data-independent hash functions is identified and is called 'RGDI hash functions.' More new hash functions will be included in the RGDI collection based on simulation results.

Shin experimented RC4 hash functions and compared their performances with those of others to make the phenomenon of RG more visible. If a well-chosen criterion is provided in a survey, the phenomenon of RG can be more observable, and this phenomenon can be seen in both a RG collection of polynomial time algorithms and that of exponential time algorithms.

As an RGDI hash function, the newly-developed Shin's hash method involves the combination of the mapping or converting of each character in a key to a corresponding prime-number or random-number technique and the folding technique. This proposed parallel processing of the Shin's hash coder transforms each character into a number and calculates each bit value in a hash address by means of hardware in order to produce a hash address within three clock cycles. Other hash methods cannot take advantage of such effective parallel processing, in producing each bit into a hash address, because of the algorithmic nature of their hash address

calculation. The Shin's hash coder in hardware is relatively inexpensive compared to other hardware hash coders, which use complex mathematical operations like multiplication and division. Compared to other well-known methods, the Shin's hash method distributes keys effectively. Moreover, it does not have a data dependency problem in its distribution of similar keys because the Shin's hash method is sensitive to every character in a key producing a hash address. Not only does it use every character in a key as input, but it also uses positions of the characters as input. Therefore, using two inputs, the Shin's hash method provides different values for a character in a key, based on the position of the character.

### 5.2 Conclusions

In this survey of hash functions, relatively good and data-independent (RGDI) hash functions have become recognized. The collection of RGDI hash functions initially included the Shin's, the additive Shin's, the Maurer's shift-fold-loading, the Berkovich's Hu-Tucker code, and Shin's RC (e.g., RC8, RC(0,10,20,30), and RC(0,11,22,25)), and the division method. The survey shows that the difference of distribution performances between the RGDI hash functions is hardly noticeable. Therefore, based on the experimental results, the phenomenon of RG is defined and discussed in this paper.

As shown in Table 1, the Shin's hash method satisfies all three requirements (i.e., distribution, speed, and cost) at the highest rank. This paper demonstrates that the Shin's hash method is better than the broadly accepted division hash method in both software and hardware implementation cases. In the software implementation case, the additive Shin's hash method is superior to the division hash method since the Shin's method divides the sum by the number of buckets (b) to produce a remainder as a hash address. In the Shin's hash method, there will be no unemployed bucket in the hash table in contrast to the division method, which divides the combination by the largest prime number smaller than b to produce a remainder as a hash address. In the division hash method, one has to use a prime number as the hash table size to avoid any waste of buckets. It is a hash method with a rigid restriction on choosing a hash table size. When a hash coder is implemented in hardware, the Shin's hash coder and Shin's RC hash coder are faster and cheaper than the division hash coder. Therefore, the (additive) Shin's hash method is recommended for all applications that use a hash function.

## ACKNOWLEDGEMENTS

We would like to thank Domenico Ferrari, David Messerschmitt, Steven Schwarz, Diogenes Angelakos, Lotfi Zadeh, Kun-Hee Lee, Michael Stonebraker, Michael Feldman, and Simon Berkovich, Helen Shin for their suggestions and feedbacks. We are grateful to Ward Maurer for his helpful advice in this survey.

## BIBLIOGRAPHY

[ABDA1] Abd-alla, A. M. and Meltzer, A. C. *Principles of Digital Computer Design.* Vol. I, Englewood Cliffs: Prentice Hall, 1976.

[AHO1] Aho, A. V., Hopcroft, J. E., and Ullman J. D. *Data Structures and Algorithms.*

Reading: Addison-Wesley, 1983.

[BABB1] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM Transactions on Database Systems,* Vol. 4, No. 1, Mar. 1979: 1-29.

[BREU1] Breuer, M. and Friedman, A. D. *Diagnosis & Reliable Design of Digital Systems.* Rockville: Computer Science Press, Inc. 1976.

[BUCH1] Buchholz, W. "File Organization and Addressing." *IBM Systems Journal,* 2, Jun. 1963: 86-111.

[CAPP1] Cappa, M. and Hamacher, V. C. "An Augmented Iterative Array for High-Speed Binary Division." *IEEE Transactions on Computers,* Vol. C-22, Feb. 1973: 172-5.

[CAVA1] Cavanagh, J. J. F. *Digital Computer Arithmetic Design and Implementation.* McGraw-Hill, 1984.

[DATE1] Date, C. J. *An Introduction to Database Systems.* Reading: Addison-Wesley, 1986.

[GARE1] Garey, M. R. and Johnson, D. S. Computers and Intractability: A Guide to the Theory of NP-Completeness. San Francisco: W. H. Freeman and Company, 1986.

[KNOT1] Knott, G. D. "Hashing functions." *The Computer Journal,* Vol. 18, No. 3, Aug. 1975: 265-78.

[KNUT1] Knuth, D. E. *The Art of Computer Programming.* Vol. 3, Sorting and Searching, Reading: Addison-Wesley, 1975.

[LUM1] Lum, V. Y., et al. "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files." *CACM,* Vol. 14, No. 4, Apr. 1971: 228-39.

[LUM2] Lum, V. Y. "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept." *CACM,* Vol. 16, No. 10, Oct. 1973: 603-12.

[MART1] Martin, J. *Computer Data-Base Organization.* 2nd Ed., Englewood Cliffs: Prentice-Hall, 1977.

[MOTO1] Motorola, Inc. *MC68030 Enhanced 32-bit Microprocessor User's Manual.*

Motorola, Inc., 1987.

[MAUR1] Maurer, W. D. "An Improved Hash Code for Scatter Storage." *CACM*, Vol. 2, No. 1, Jan. 1968: 35-8.

[MAUR2] Maurer, W. D. and Lewis, T. G. "Hash Table Methods." *ACM's Computing Surveys*, Vol. 7, No. 1, Mar. 1975: 5-19.

[MORR1] Morris, R. "Scatter Storage Techniques." *CACM*, Vol. 2, No. 1, Jan. 1968: 38-44.

[PEAR1] Pearson, P. K. "Fast Hashing of Variable-Length Text Strings." *CACM*, Vol. 33, No. 6, Jun. 1990: 677-80.

[RAMA1] Ramakrishna, M. V. "Hashing in Practice, Analysis of Hashing and Universal Hashing." *Proceedings of ACM's SIGMOD 1988 International Conference on Management of Data*, Vol. 17, No. 3, Sep 1988: 191-99.

[SHIN1] Shin, D. K. *A Comparative Study of Hash Functions for a New Hash-Based Relational Join Algorithm.* Pub. #91-23423, Ann Arbor: UMI Dissertation Information Service, 1991.

[SHIN2] Shin, D. K. and Meltzer, A. C. "A Comparative Study of Hash Functions for an Effective Hash Coder." which was submitted to *Communications of the ACM* MS No. A869, in Aug. 1991.

[SHIN3] Shin, D. K. and Meltzer, A. C. "A New Join Algorithm." *ACM SIGMOD RECORD*, Vol. 23, No. 4, Dec. 1994: 13-8.

[STEF1] Stefanelli R. "A Suggestion for a High-Speed Parallel Binary Divider." *IEEE Transactions on Computers*, Vol. C-21, No. 1, Jan. 1972: 113-9.

[ULLM1] Ullman, J. D. *Principles of Database Systems.* Rockville: Computer Science Press, 1982.

[WALL1] Wallace, C. S. "A Suggestion for a Fast Multiplier." *IEEE Transactions on Electronic Computers*, Vol. EC-13, No. 1, Feb. 1964: 14-7.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 87 | 49 | 12 | 176 | 178 | 102 | 166 |
| 121 | 193 | 6 | 84 | 249 | 230 | 44 | 163 |
| 14 | 197 | 213 | 181 | 161 | 85 | 218 | 80 |
| 64 | 239 | 24 | 226 | 236 | 142 | 38 | 200 |
| 110 | 177 | 104 | 103 | 141 | 253 | 255 | 50 |
| 77 | 101 | 81 | 18 | 45 | 96 | 31 | 222 |
| 25 | 107 | 190 | 70 | 86 | 237 | 240 | 34 |
| 72 | 242 | 20 | 214 | 244 | 227 | 149 | 235 |
| 97 | 234 | 57 | 22 | 60 | 250 | 82 | 175 |
| 208 | 5 | 127 | 199 | 111 | 62 | 135 | 248 |
| 174 | 169 | 211 | 58 | 66 | 154 | 106 | 195 |
| 245 | 171 | 17 | 187 | 182 | 179 | 0 | 243 |
| 132 | 56 | 148 | 75 | 128 | 133 | 158 | 100 |
| 130 | 126 | 91 | 13 | 153 | 246 | 216 | 219 |
| 119 | 68 | 223 | 78 | 83 | 88 | 201 | 99 |
| 122 | 11 | 92 | 32 | 136 | 114 | 52 | 10 |
| 138 | 30 | 48 | 183 | 156 | 35 | 61 | 26 |
| 143 | 74 | 251 | 94 | 129 | 162 | 63 | 152 |
| 170 | 7 | 115 | 167 | 241 | 206 | 3 | 150 |
| 55 | 59 | 151 | 220 | 90 | 53 | 23 | 131 |
| 125 | 173 | 15 | 238 | 79 | 95 | 89 | 16 |
| 105 | 137 | 225 | 224 | 217 | 160 | 37 | 123 |
| 118 | 73 | 2 | 157 | 46 | 116 | 9 | 145 |
| 134 | 228 | 207 | 212 | 202 | 215 | 69 | 229 |
| 27 | 188 | 67 | 124 | 168 | 252 | 42 | 4 |
| 29 | 108 | 21 | 247 | 19 | 205 | 39 | 203 |
| 233 | 40 | 186 | 147 | 198 | 192 | 155 | 33 |
| 164 | 191 | 98 | 204 | 165 | 180 | 117 | 76 |
| 140 | 36 | 210 | 172 | 41 | 54 | 159 | 8 |
| 185 | 232 | 113 | 196 | 231 | 47 | 146 | 120 |
| 51 | 65 | 28 | 144 | 254 | 221 | 93 | 189 |
| 194 | 139 | 112 | 43 | 71 | 109 | 184 | 209 |

Figure 1. Pearson's Auxiliary Table T

```
procedure Pearson_Hash (var Hash_Value : integer);
var
        i : integer;
begin
        H(0) := 0;
        for i := 1 to NUM_CHARS_IN_KEY do
          begin
              H(i) := T(Exclusive_Or (H(i - 1), C(i)));
          end;
        Hash_Value := H(NUM_CHARS_IN_KEY);
        end;
```

Figure 2. Pearson's Hash Algorithm

```
N := N Exclusive_OR (Rotate_Right (N, 1))
N := N Exclusive_OR (Rotate_Right (N, 3))
N := N Exclusive_OR (Rotate_Right (N, 7))
N := N Exclusive_OR (Rotate_Right (N, 15))
N := N Exclusive_OR (Rotate_Right (N, 31))
N := N Exclusive_OR (Rotate_Right (N, 63))
N := N Exclusive_OR (Rotate_Right (N, 127))
```

where the statement "N := N Exclusive_OR (Rotate_Right (N, K))" assigns the resulting value from exclusive-OR of both intermittent value (N) and K bits rotated value of N back to the intermittent value.

Figure 3. The Algorithm of Maurer's Shift-fold-loading

| | | | |
|---|---|---|---|
| SPACE: | 000 | n, N : | 1010 |
| a, A : | 0010 | o, O : | 1011 |
| b, B : | 001100 | p, P : | 110000 |
| c, C : | 001101 | q, Q : | 110001 |
| d, D : | 00111 | r, R : | 11001 |
| e, E : | 010 | s, S : | 1101 |
| g, G : | 01101 | t, T : | 1110 |
| h, H : | 0111 | u, U : | 111100 |
| i, I : | 1000 | v, V : | 111101 |
| j, J : | 1001000 | w, W : | 111110 |
| k, K : | 1001001 | x, X : | 11111100 |
| l, L : | 100101 | y, Y : | 11111101 |
| m, M : | 10011 | z, Z : | 1111111 |

Figure 4. Hu-Tucker Codes [KNUT1]

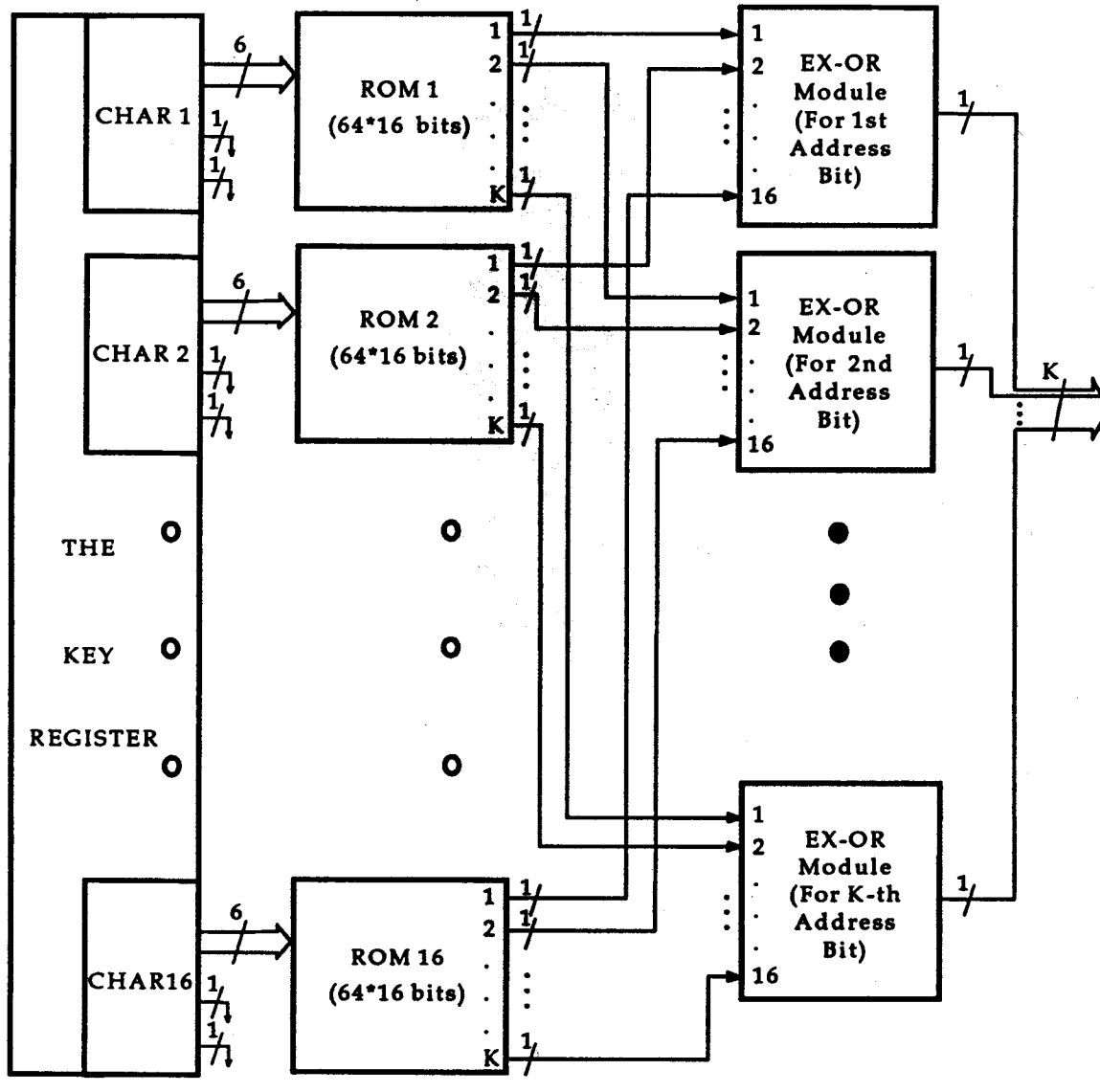


Figure 5. Shin's Hash Coder

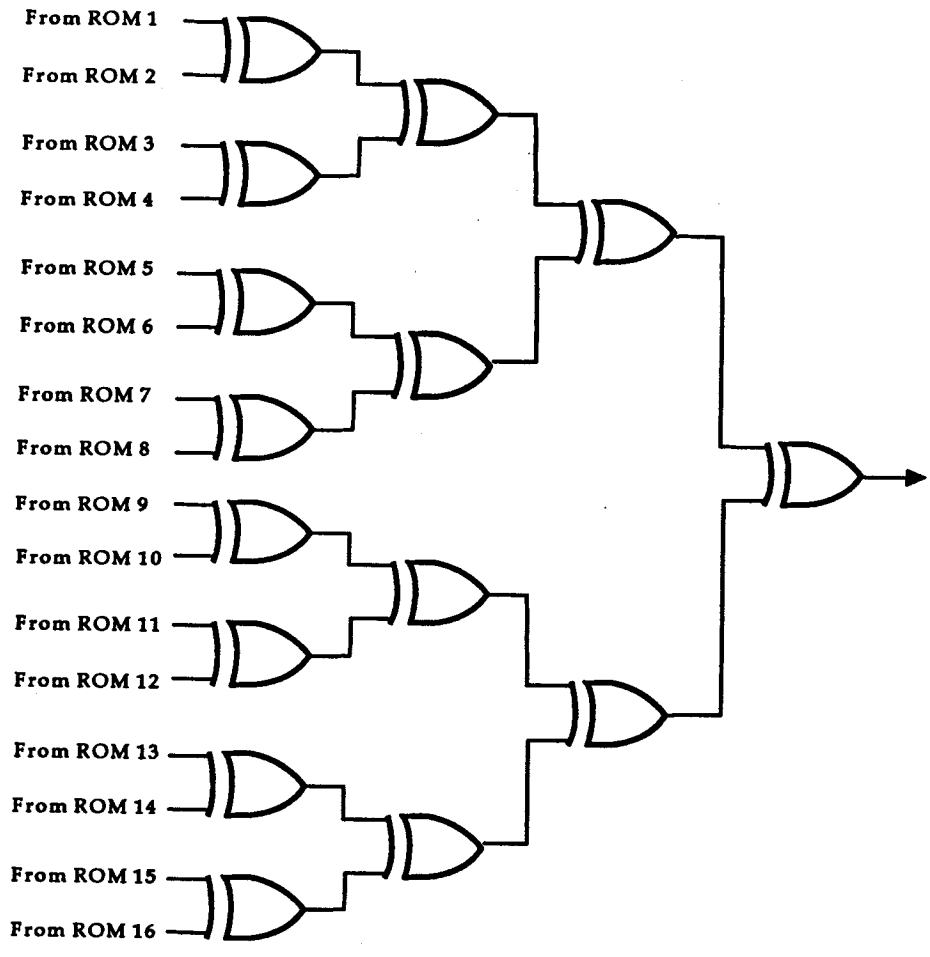


Figure 6. A EX-OR Module

```
const
        MAX_NO_CHARS_IN_KEY = 16; {number of characters in a key}
        MAX_NO_BUCKETS = 256; {number of buckets in the hash table}
        NO_PRIMES_IN_ROM = 64; {number of prime numbers in each ROM}

type
        {Type for the array of 16 characters key}
        Key_Array_Type = array [1..MAX_NO_CHARS_IN_KEY] of char;

var
        {Array table of 64 prime numbers for each ROM}
        Prime_Table : array [1..MAX_NO_CHARS_IN_KEY, 0..NO_PRIMES_IN_ROM-1]
                        of integer;

function Shin_Hash (Key : Key_Array_Type) : integer;
var
        Temp, Char_No, Index : integer;
begin
        Temp := 0;
        for Char_No := 1 to MAX_NO_CHARS_IN_KEY do
        begin
                Index := ord(Key[Char_No]);
                if Index >= NO_PRIMES_IN_ROM then
                        Index := Index - NO_PRIMES_IN_ROM;
                Temp := EX_OR(Prime_Table[Char_No,Index], Temp);
        end;
        Mapping_Hash := Temp mod MAX_NO_BUCKETS;
end;
```

Figure 7. Shin's Hash Algorithm

```
const
        MAX_NO_CHARS_IN_KEY = 16; {number of characters in a key}
        MAX_NO_BUCKETS = 256; {number of buckets in the hash table}
        NO_PRIMES_IN_ROM = 128; {number of prime numbers in each ROM}

type
        {Type for the array of 16 characters key}
        Key_Array_Type = array [1..MAX_NO_CHARS_IN_KEY] of char;

var
        {Array table of 64 prime numbers for each ROM}
        Prime_Table : array [1..MAX_NO_CHARS_IN_KEY, 0..NO_PRIMES_IN_ROM-1]
                        of integer;

function Additive_Shin_Hash (Key : Key_Array_Type) : integer;
var
        Temp, Char_No, Index : integer;
begin
        Temp := 0;
        for Char_No := 1 to MAX_NO_CHARS_IN_KEY do
        begin
                Index := ord(Key[Char_No]);   {Index will be a number between 0 and 127}
                Temp  := Prime_Table[Char_No,Index] + Temp;
        end;
        Mapping_Hash := Temp mod MAX_NO_BUCKETS;
end;
```

Figure 8. Additive Shin's Hash Algorithm

```
function A_Prime_Number (number : integer) : boolean;
{This function will return true if the input number is a prime number or return false otherwise.}
var
        i, max_factor : integer;
        finish : boolean;
begin
        if (number mod 2) = 0 then  {If the input number is a even number, return false.}
                A_Prime_Number := false
        else begin

                {Produce a number which can be a possible maximum factor of the input.}
                max_factor := round (sqrt(number));

                finish := false;
                i := 3;
                while (i <= max_factor) and (not finish) do
                begin
                        if (number mod i) = 0 then
                                finish := true      {It's not a prime number, so stop looping}
                        else
                                i := i + 2;
                end;
                if finish then
                        A_Prime_Number := false
                else
                        A_Prime_Number := true;
        end;
end;


Procedure Finding_Prime_Numbers (first_num : integer; last_num : integer);
{find prime numbers with a range specified in input parameters.}
var
        index : integer;

begin
        for index := first_num to last_num do  {find prime numbers within a range}
                if A_Prime_Number (index) then
                        writeln(index);
end;
```

Figure 9. Prime Number Finding Algorithm

| Hash Method | <Distribution> MSD When Applied to RCN | GCN | RNS | <Speed> Clock Cycles SW | HW | <Cost> No. of Gates |
|---|---|---|---|---|---|---|
| Shin's Mapping | 3.93 Avr. | 4.06 Avr. | 4.07 Avr. | 96 | 3 (1) | 120 (2) |
| Shin's Additive Mapping | 4.40 | 3.91 | 3.58 | 96 | 64 | 182 |
| Maurer's Shift-fold-loading | 3.95 Avr. | 4.06 Avr. | 3.81 Avr. | 420 | 70 | 384 |
| Berkovich's Hu-Tucker Code | 4.09 | 3.97 | 3.70 | 826 (3) | 128 (3) | 399 |
| Shin's<br>FS(0,10,20,30)<br>FS(0,11,22,25) | <br>4.20<br>4.03 | <br>3.96<br>4.46 | <br>4.27<br>4.88 | <br>44 | <br>1 (4) | <br>192 |
| Division (Divisor = 241) | 5.51 | 5.35 | 4.48 | 70 | 46<br>16 (5) | 390<br>3360 (5) |
| Pearson's Table Indexing | 20.63 | 21.23 | 21.15 | 82 | 82 | 280 |
| Digit Analysis<br>(2 and 4 bytes) | 4.32<br>3.80 | 4.07<br>4.70 | 3.84<br>19.74 | 40 (6) | 2 (6) | 112<br>96 |
| Fold-boundary | 4.09 | 3.89 | 53.02 | 56 | 1 (4) | 117 |
| Midsquare | 4.25 | 4.84 | 88.91 | 72 | 30<br>8 (7) | 572<br>2796 (7) |
| Multiplicative | 4.42 | 3.29 | 12.49 | 407 | 64<br>17 (7) | 422<br>2892 (7) |
| Radix | 3.97 | 4.05 | 12.36 | 650 | 390<br>285 (7)<br>120 (8) | 550<br>3234 (7)<br>6498 (8) |
| Random | 4.25 | 3.63 | 9.79 | 162 | 80<br>57 (7)<br>26 (8) | 470<br>3138 (7)<br>6402 (8) |

(1) Calculation time is measured after a key is stored in the key register.
(2) Sixteen 64*16 bits ROMs are also required.
(3) Changeable due to variable length encoded key string.
(4) Calculation time is measured after an encoded key is stored in the key register.
(5) Faster but expensive since special hardware (division array [CAPP1, STEF1]) is used.
(6) Analysis for digits is required beforehand.
(7) Faster but expensive due to Wallace Tree [WALL1] for multiplication.
(8) Both Wallace Tree and division array are used.

Table 1. Performances of Hash Methods

| The Mean Square Deviations of the Mapping Hash Method Using Prime Numbers | | | | | | |
|---|---|---|---|---|---|---|
| | Selected Bits for a Hash Address | | | | | |
| Data Set | 2-9 | 3-10 | 4-11 | 5-12 | 6-13 | Average |
| RCN | 3.74 | 3.83 | 4.13 | 4.20 | 3.77 | 3.93 |
| GCN | 4.41 | 4.54 | 3.91 | 3.83 | 3.60 | 4.06 |
| RNS | 3.95 | 4.05 | 4.40 | 3.74 | 4.22 | 4.07 |

Table 2A. Distribution Performance of the Mapping Hash Method with Prime Numbers

| The Mean Square Deviations of the Mapping Hash Method Using Random Numbers | | | | | | |
|---|---|---|---|---|---|---|
| | Selected Bits for a Hash Address | | | | | |
| Data Set | 1-8 | 2-9 | 3-10 | 4-11 | 5-12 | Average |
| RCN | 3.95 | 3.72 | 4.69 | 4.06 | 4.20 | 4.12 |
| GCN | 3.48 | 3.63 | 3.87 | 3.89 | 3.70 | 3.71 |
| RNS | 3.77 | 3.91 | 4.03 | 4.48 | 4.16 | 4.07 |

Table 2B. Distribution Performance of the Mapping Hash Method with Random Numbers

| Shin's FS Hash Function | Data Sets | <Selected Bits> 1-8 | 9-16 | 17-24 | 25-32 |
|---|---|---|---|---|---|
| Shin's FS(0,10,20,30) hash function | RCN | 4.48 | 3.88 | 4.20 | 3.84 |
| | GCN | 4.77 | 3.63 | 3.96 | 4.27 |
| | RNS | 3.44 | 4.58 | 4.41 | 3.56 |
| R(0,2,4,6) => FS(0,2+8*1,4+8*2,6+8*3) = FS(0,10,20,30) | | | | | |
| | Data Sets | <Selected Bits> 1-8 | 9-16 | 17-24 | 25-32 |
| Shin's FS(0,11,22,25) hash function | RCN | 3.73 | 4.23 | 4.03 | 3.34 |
| | GCN | 4.34 | 3.71 | 4.46 | 4.14 |
| | RNS | 3.51 | 4.19 | 4.88 | 3.94 |
| R(0,3,6,1) => FS(0,3+8*1,6+8*2,1+8*3) = FS(0,11,22,25) | | | | | |
| | Data Sets | <Selected Bits> 1-8 | 9-16 | 17-24 | 25-32 |
| Shin's FS(0,12,17,29) hash function | RCN | 4.23 | 4.12 | 4.41 | 4.02 |
| | GCN | 4.43 | 4.15 | 4.98 | 3.74 |
| | RNS | 4.02 | 4.98 | 20.70 | 3.69 |
| R(0,4,1,5) => FS(0,4+8*1,1+8*2,5+8*3) = FS(0,12,17,29) | | | | | |
| | Data Sets | <Selected Bits> 1-8 | 9-16 | 17-24 | 25-32 |
| Shin's FS(0,13,18,31) hash function | RCN | 3.80 | 3.84 | 4.46 | 3.83 |
| | GCN | 3.55 | 4.33 | 5.09 | 3.63 |
| | RNS | 20.05 | 21.16 | 20.13 | 4.16 |
| R(0,5,2,7) => FS(0,5+8*1,2+8*2,7+8*3) = FS(0,13,18,31) | | | | | |
| | Data Sets | <Selected Bits> 1-8 | 9-16 | 17-24 | 25-32 |
| Shin's FS(0,15,22,29) hash function | RCN | 4.07 | 4.11 | 4.05 | 4.42 |
| | GCN | 4.17 | 4.13 | 5.28 | 4.05 |
| | RNS | 53.02 | 20.64 | 21.60 | 21.00 |
| R(0,7,6,5) => FS(0,7+8*1,6+8*2,5+8*3) = FS(0,15,22,29) | | | | | |

Table 3. Distribution Performances of Shin's Various FS (Fold-Shifting) Hash Methods

| The Mean Sqaure Deviations of the Division Method | | | |
|---|---|---|---|
| Divisor Used | RCN | GCN | RNS |
| 241 (1) | 5.51 | 5.35 | 4.48 |
| 242 | 4.94 | 5.34 | 21.91 |
| 243 | 4.43 | 4.52 | 4.98 |
| 244 | 4.78 | 4.80 | 21.00 |
| 245 | 4.19 | 5.39 | 4.95 |
| 246 | 4.48 | 3.94 | 21.02 |
| 247 | 4.28 | 4.67 | 4.49 |
| 248 | 4.15 | 4.56 | 20.51 |
| 249 | 4.29 | 5.66 | 4.21 |
| 250 | 4.90 | 4.66 | 20.73 |
| 251 | 3.80 | 4.30 | 4.34 |
| 252 | 4.55 | 4.24 | 20.48 |
| 253 | 4.09 | 4.84 | 4.92 |
| 254 | 3.95 | 4.12 | 93.72 |
| 255 | 5.77 | 8.23 | 107.82 |
| 256 (2) | 25.63 | 20.20 | 502.54 |
| 257 | 5.67 | 11.95 | 122.99 |
| 258 | 4.59 | 4.45 | 93.13 |
| 259 | 4.85 | 6.60 | 4.10 |
| 260 | 5.07 | 5.11 | 20.28 |
| 261 | 3.13 | 4.99 | 3.95 |
| 262 | 4.58 | 3.51 | 21.38 |
| 263 (1) | 4.71 | 4.31 | 4.68 |

(1) Prime Number Divisor --- 263 and 241
(2) Number of Buckets --- 256

Table 4. Distribution Performance of the Division Hash Method

| The Mean Square Deviations of the Fold-boundary Method | | | |
|---|---|---|---|
| Bits Selected | RCN | GCN | RNS |
| 11, 12, ..., 18 | 4.09 | 3.89 | 53.02 |
| 12, 13, ..., 19 | 3.72 | 3.89 | 53.86 |
| 13, 14, ..., 20 | 3.63 | 3.62 | 56.23 |

Table 5. Distribution Performance of the Fold-boundary Hash Method

| The Mean Square Deviations of the Midsquare Method | | | |
|---|---|---|---|
| Bits Selected | RCN | GCN | RNS |
| 11, 12, ..., 18 | 4.45 | 4.48 | 68.55 |
| 12, 13, ..., 19 | 4.76 | 5.13 | 72.52 |
| 13, 14, ..., 20 | 4.25 | 4.84 | 88.91 |

Table 6. Distribution Performance of the Midsquare Hash Method

| The Mean Square Deviations of the Shift-fold-loading method | | | |
|---|---|---|---|
| Bits Selected | RCN | GCN | RNS |
| 10, 11, ..., 17 | 4.29 | 4.13 | 3.88 |
| 20, 21, ..., 27 | 3.27 | 3.84 | 4.04 |
| 30, 31, ..., 37 | 3.79 | 4.66 | 4.45 |
| 40, 41, ..., 47 | 4.13 | 4.11 | 4.07 |
| 50, 51, ..., 57 | 3.66 | 3.71 | 3.81 |
| 60, 61, ..., 67 | 3.74 | 4.41 | 3.95 |
| 70, 71, ..., 77 | 3.83 | 4.54 | 4.05 |
| 80, 81, ..., 87 | 4.13 | 3.91 | 4.40 |
| 90, 91, ..., 97 | 4.20 | 3.83 | 3.74 |
| 100, 101, ...,107 | 3.77 | 3.60 | 4.22 |
| Average: | 3.95 | 4.12 | 3.97 |

Table 7. Distribution Performance of Maurer's Shift-fold-loading Hash Method