A Collection of Research Processes for Genealogy and Proofs

VOLUME THIRTY, SECTION 256

Submissions and Some Papers Revised in Previous Years (1994-1995)

by

Dr. Dong-Keun Shin

March 1998

Submitted to the Chair of
Department of Electrical Engineering and Computer Sciences
College of Engineering
University of California, Berkeley
Berkeley, CA 94720
U. S. A.

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

1210 WEST DAYTON STREET
MADISON, WISCONSIN 53706
TELEPHONE (608) 262-2252

October 11, 1994

Mr. Dong Keun Shin
Kyungki-Do Pundang-Gu
Suhyun-Dong 299-Bunji
Hyundai Apt. 112-502
South Korea

Dear Mr. Shin:

This is to acknowledge receipt of the paper **A New Efficient Hash-Based Relational Join Algorithm and HIMOD-A Database Computer** which you have submitted to the *1995 ACM SIGMOD International Conference on Management of Data (SIGMOD-95)*. The SIGMOD-95 program committee is currently scheduled to meet in mid-January of 1995, and we will notify you of the committee's decision by January 20th at the latest.

Thank you for submitting your paper to SIGMOD-95.

Sincerely,

Michael J. Carey
SIGMOD-95 Program Chair

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

1210 WEST DAYTON STREET
MADISON, WISCONSIN 53706
TELEPHONE (608) 262-2252

October 11, 1994

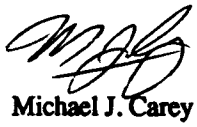
Mr. Dong Keun Shin
Kyungki-Do Pundang-Gu
Suhyun-Dong 299-Bunji
Hyundai Apt. 112-502
South Korea

Dear Mr. Shin:

This is to acknowledge receipt of the paper **A Survey of Hash Functions for an Effective Hash Coder** which you have submitted to the *1995 ACM SIGMOD International Conference on Management of Data (SIGMOD-95)*. The SIGMOD-95 program committee is currently scheduled to meet in mid-January of 1995, and we will notify you of the committee's decision by January 20th at the latest.

Thank you for submitting your paper to SIGMOD-95.

Sincerely,



Michael J. Carey
SIGMOD-95 Program Chair

COMPUTER SCIENCES DEPARTMENT
UNIVERSITY OF WISCONSIN-MADISON

1210 WEST DAYTON STREET
MADISON, WISCONSIN 53706
TELEPHONE (608) 262-2252

October 13, 1994

Mr. Dong Keun Shin
Kyungki-Do Pundang-Gu
Suhyun-Dong 299-Bunji
Hyundai Apt. 112-502
SOUTH KOREA

Dear Mr. Shin:

This is to acknowledge receipt of the paper **A New Efficient Relational Join Algorithm and HIMOD-A Database Computer** which you have submitted to the *1995 ACM SIGMOD International Conference on Management of Data (SIGMOD-95)*. The SIGMOD-95 program committee is currently scheduled to meet in mid-January of 1995, and we will notify you of the committee's decision by January 20th at the latest.

Thank you for submitting your paper to SIGMOD-95.

Sincerely,

Michael J. Carey
SIGMOD-95 Program Chair

From jag@research.att.com Fri Jun 10 23:50:06 1994
Received: from research.att.com ([192.20.225.3]) by saitgw.sait.samsung.co.kr (4.1/SMI-4.1)
	id AA11194; Fri, 10 Jun 94 23:50:23 KST
Message-Id: <9406101450.AA11194@saitgw.sait.samsung.co.kr>
Received: by research.att.com; Fri Jun 10 10:44 EDT 1994
Received: from vidya.tempo.att.com.UUCP by allegra.tempo.att.com; id AA21160; Fri, 10 Jun 94 10:44:19 EDT
From: jag@research.att.com (H. V. Jagadish)
Date: Fri, 10 Jun 1994 10:44:16 EDT
X-Mailer: Mail User's Shell (7.1.1 5/02/90)
To: dkshin@trvax5.sait.samsung.co.kr
Subject: Your manuscripts
Status: R

I taook a quick look at thr manuscripts you sent me.

The survey paper is interesting, but publishable only in ACM Computing Surveys or other forums that accept surveys. Most confernces and journals will require original work.

The new hash-join algorithm paper suffers from two drawbacks --
First, it is best to present your algorithm divorced from specific hardware implementation. It is generally accepted today that specialized hardware is not viable.
Second, be fair in your comparisons. 200 times better than nested-loops is an impressive-sounding statement with zero information. You should compare against the best known competition, for instance, hybrid hash join. Even if you can only claim that you are 20% better, and only in some operating regions, that will be more impressive than what you currently have.

Best of luck.

'200 times better ...' will be eliminated from the abstract part of my join algorithm paper.

All of the hash joins including my join algorithm require O(n) operation. However, my hash join algorithm is more efficient since it eliminates unnecessary data as early as possible using the divide and conquer strategy. Therefore, I believe that it will outperform other hash joins. (People know that the quicksort outperforms the mergesort, the heapsort, and the shellsort although all of them have O(NlogN) performance since the quicksort efficiently sorts data using the divide and conquer strategy.)

Specialized hardware is not mandatory for realization of my join algorithm. This statement will be added in the paper.

D. K. Shin, D.Sc.

A SURVEY OF HASH FUNCTIONS FOR AN EFFECTIVE HASH CODER

Dong Keun Shin and Arnold Charles Meltzer

Samsung Electronics Co., Ltd.
Communication Systems R&D Center
Songpa P.O.Box 117, Seoul, Korea
E-mail: dkshin@telecom.samsung.co.kr

Department of Electrical Engineering and Computer Science
The School of Engineering and Applied Science
The George Washington University
E-mail: meltzer@seas.gwu.edu

Abstract

Shin's mapping hash method and several Shin's FS hash methods are designed to take advantage of parallel processing in calculating each bit value of a hash address. The new mapping hash method not only has reliable and relatively good key distribution, but it also takes only a few clock cycles to calculate a hash address if the mapping hash coder is implemented in hardware.

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 was based on criteria such as distribution, speed, and cost. As a result of this study, a group of relatively good and data-independent (RGDI) hash functions are recognized. No noticeable difference has been found in distribution performances of hash functions in the RGDI group. Among the RGDI hash functions, the Shin's mapping 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 (additive) mapping 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 [MAUR2, BABB1]; therefore, there has been a huge demand for an effective hash coder. In some applications, an effective hash coder is essential to increase the speed of hash-based operations. Another motive for finding a good hash function and the survey of hash functions is that performance of some application is heavily dependent on distribution performance of a chosen hash function. 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 application [BABB1, SHIN2] since the number of keys the hash coder has to transform into hash addresses may be large. The hash address calculation per each key often is 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 in 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].

According to this survey of hash functions, 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 of a collision occurring. 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, as 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 performance. 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, Berkovich's Hu-Tucker code, Shin's mapping and additive mapping, and Shin's various versions of FS (fold-shifting) are introduced and compared with currently existing hash functions in terms of distribution, speed, and cost. In Section 2, 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. Finally, 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, random, and Shin's FS. On the other hand, hash functions such as Shin's mapping, Maurer's shift-fold-loading, Berkovich's Hu-Tucker code, and Pearson's table indexing 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 hashing function can be directly applied. As Maurer suggests [MAUR1], 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 is selected. Each hash method is executed on the three data sets to produce mean square deviations, the better the distribution and the fewer incidences of collision. Since the number of buckets in the hash table is 256, and since 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))

Two speed performances should be determined: one for a software implementation and the other for a hardware implementation. First, when a hash function if 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

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

instruction-cache case times listed in the tables of the MC68030 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 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, MAUR2].

3.2 The Digit Analysis Hash Method

The digit analysis hash method [MAUR1, 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 [MAUR1, 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. New versions of fold-shifting (FS) are developed and discussed in this paper.

3.4 The Midsquare Hash Method

In the midsquare hash method [MAUR1, 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 [MAUR2, 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 [MAUR1, LUM2], a number representing the key is considered as a 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 like 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 [MAUR1] 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 which 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 that 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 the 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 the bits in the shift register are rotated three bits right, and exclusive-ORing and loading follows 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.

The idea 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 Mapping Hash Method

The algorithm of Shin's mapping hash method in a Pascal-like notation is shown in Figure 5. 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 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. Then the mapping hash function converts the resulting binary string back to integer output to be sent to the calling program. In the hardware implementation of this mapping hash method, the exclusive-OR operation is more valuable than the addition operation since 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, Shin's additive mapping hash function which employs the addition operation can be used instead.

With the last statement, the time-consuming MOD operation that provides a remainder after a division, is not necessary if the least significant **k** bits from the combination or the sum can be extracted in order to produce a hash address for the table of 2**k buckets. This alternative method is acceptable, since the value (the combination or the sum) in the variable Temp is already adequately randomized.

The mapping hash function is actually a hardware-oriented hash method that converts or maps in parallel the internal representation, e.g., ASCII code, of each character in a key to an arbitrarily chosen prime number (or a randomly chosen number), and 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 of two to the power of **k** buckets. The 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 6 and 7. In each ROM, 64 (2**6) the 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.) All the selected numbers should be greater than the number of buckets in a hash table. A set of all 16 ROMs is included in the hardware mapping hash coder. 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 6, 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 a hash address also are constructed at the same time. The circuit of EX-OR Module for each bit shown in Figure 6 is represented in Figure 7. The concurrent processing in looking up random numbers and in 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 in 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. One may add one to every prime numbers in even number slots in each ROM to avoid the problem.

The following statement in the algorithm, "Temp := EX_OR (Prime_Table (Index), Temp);" performs exactly the same function that the hardware implemented mapping 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:

$$(((X1+X2)+(X3+X4))+((X5+X6)+(X7+X8)))+$$
$$(((X9+X10)+(X11+X12))+((X13+X14)+(X15+X16)))$$

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

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 the associative law of exclusive-OR, such that $(X+Y)+Z = X+(Y+Z) = X+Y+Z$, one can easily prove both sides of the above equation are equal. Thus, it is obvious that the parallel and serial processing of the mapping hash method are equivalent. Considering limited bandwidth in transferring keys, character by character serial processing for hashing a key is also feasible using an iterative

hardware component. By the law of the exclusive-OR, the parallel processing and the serial processing of the hardware mapping hash method provide the same hash address calculation.

3.12 Shin's Additive Mapping Hash Method

The algorithm of Shin's additive mapping hash method employs the addition operation instead of the exclusive-OR operation that the Shin's mapping hash method employs. When one implement the Shin's additive mapping 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, the sum in the variable Temp is 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 mapping hash method, as shown in this survey.

3.13 Shin's FS (Fold-Shifting) Hash Method

As has been shown by several researchers [MAUR1, KNUT1, 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 which are shifted in their connection to exclusive-OR gates.

When there is an encoded one word key (or partition), there may be many ways to fold using the exclusive-OR operation. The questions about the fold-shifting method can be described 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 R(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, R(0,2,4,6) is equivalent to any combination of 0,2,4, and 6, e.g., R(2,4,6,0), R(4,6,0,2), etc. R(0,2,4,6) also is symmetric to R(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, R(0,3,6,1) would be considered. If the number

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

These fold-shifting hash methods may require that the number of rotated keywords should 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, R(0,2,4,6) becomes FS(0,2+8, 4+8*2, 6+8*3) or FS(0,10,20,30). By the same process, R(0,3,6,1) becomes FS(0,11,22,25), R(0,4,1,5) becomes FS(0,10,17,29), R(0,5,2,7) becomes FS(0,13,18,31), and R(0,7,6,5) becomes FS(0,15,22,29). Therefore, the selected FS hash methods to be examined are FS(0,10,20,30), FS(0,11,22,25), FS(0,10,17,29), FS(0,13,18,31), and FS(0,10,17,29). The distribution performances of the FS hash methods are discussed in this survey.

4. 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. For 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 the measurement of the speeds of the hash coders. The cost of building a hardware hash coder is roughly represented according to the number of gates needed.

Distribution performances of the mapping 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 mapping 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 mapping 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 mapping hash method requires as many clock cycles as other hash methods require to load a key string into the key register. Two clock cycles of the MC68030 processor are required in order for the memory read to retrieve a random number from the corresponding ROM, as is specified in the Motorola's users manual [MOTO1]. One clock cycle is needed for the calculation process for 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 mapping hash coder calculates a hash address in its unique way. The hash addresses generated by different mapping 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 statistical independence that constitutes the asset of the mapping hash function. This property also is valuable in an application environment which uses rehashing scheme. It is noteworthy that the additive mapping 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 distribution performances of Shin's FS hash method, in particular, FS(0,10,20,30) and FS(0,11,22,25) are as good as those of other acceptable hash methods. But other selected FS methods, such as FS(0,12,17,29), FS(0,13,18,31), and FS(0,15,22,29), show a data dependency problem, such that 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 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 a hash table will be empty. Therefore, 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 distribution performance of the division hash method [BUCH1, MAUR1, 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 which are greater than the number of buckets in a hash table (i.e., 256) also are tested. 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 matched hash address which is within the range of the hash table. As recommended by Lum and his colleagues, the divisor 257 is a nonprime number with prime factors less than 20, but it shows very poor distributions (MSDs of 5.67, 11.95, and 122.99). As Maurer and Buchholz suggested [MAUR2, BUCH1], using the largest prime number, (i.e., 241) that also is smaller than the number of buckets, as the divisor, yields better results (MSDs of 5.51, 5.35, 4.48).

We disapprove of the division hash method [AHO1] which sums the integers (ASCII values) for each character and divides the result by the number of buckets(B), taking the remainder, which is an integer from 0 from B-1. In this case, the addressing range of the division hash method is very limited. Accordingly, this division hash method results in performing a poor distribution when the number of buckets in a hash table is larger than the range. 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 since either the mapping or the additive mapping method can be used instead, depending on the application environment. In the application, where fast hash address calculation is not required, the additive mapping method is superior to the division method. When using the additive mapping method, one need not worry about selecting a correct divisor; one need only 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 mapping hash coder which is faster and cheaper than the division hash coder is thus recommended.

Pearson's table indexing hash method appears to be erratic owing to its poor distribution performance. The fold-boundary and the midsquare show data dependency problems as shown 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. The 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.

The 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 also are reflected in the costs of a hash coder in order to help a computer designer make the best decision.

As result of the survey, a group of relatively good and data-independent hash functions is recognized, the mapping, the additive mapping, the shift-fold-loading, the Hu-Tucker code, FS(0,10,20,30) and FS(0,11,22,25), and the division are called RGDI (pronounce it like rugdy) hash functions. It is 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 the 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.,"0100101001", "1110100110", etc.) after a encoding process, all bit values in an encoded word will be fixed (i.e., stuck)except four least significant bits for four bytes in the encoded word. The performance of the division hash method must be very poor in this case. Therefore, the hash function which 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).

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. The ideal hash function design for this application is thus 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 mapping, additive mapping, shift-fold-loading, Hu-Tucker code, and various versions of FS, are 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 above tables 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 FS hash methods are reliable in terms of distribution performance; nonetheless, these methods are extremely fast in that it requires only a single clock cycle after the key is encoded and loaded into the key register. Moreover, the FS hash coder is inexpensive. This survey also shows that there is no distinguishable difference between distribution performances of relatively good, data-independent hash functions, such as the mapping, the additive mapping, the shift-fold-loading, the Hu-Tucker code, FS(0,10,20,30) and FS(0,11,22,25), and the division methods. In this survey of hash methods, the difference in the distribution performances of the relatively good hash functions is hardly perceptible. By comparing distribution performance of a hash function with the performance of other acceptable hash functions, one can easily discern whether the distribution performance of a hash function is in a group of the relatively good hash functions or not. This study group the hash functions together according to distribution performance. As a result of the survey of hash functions, a group of the relatively good and data-independent hash functions is identified, called RGDI (pronounced like rugdy) hash functions. More new hash functions will be included in the RGDI group based on simulation results.

As a RGDI hash function, the newly-developed mapping 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 mapping 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 in a hash address due to the algorithmic nature of their hash address calculation. The mapping 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, this mapping hash method distributes keys effectively. Moreover, it does not have a data dependency problem in its distribution of similar keys because the mapping hash method is sensitive to every character in a key producing a hash address.

5.2 Conclusions

In this survey of hash functions, a group of relatively good and data-independent (RGDI) hash functions is recognized. The RGDI group initially includes the mapping, the additive mapping, the shift-fold-loading, the Hu-Tucker code, FS(0,10,20,30) and FS(0,11,22,25), and the division method.

As shown in Table 1, the Shin's mapping hash method satisfies all three requirements (i.e., distribution, speed, and cost) at the highest rank. This paper demonstrates that the Shin's mapping hash method is better than the broadly accepted division hash method in both software and hardware implementation cases. In software implementation case, the Shin's additive mapping hash method is superior to the division hash method. 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 mapping 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 a 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 mapping hash coder is faster and cheaper than the division hash coder. Therefore, the Shin's (additive) mapping hash method is recommended for all the applications that use a hash function.

ACKNOWLEDGEMENTS

We would like to thank Domenico Ferrari, Michael Stonebraker, Michael Feldman, and Simon Berkovich for their suggestions. We are grateful to Ward Maurer for his helpful directions 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.

[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.

[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.

[MOTO1] Motorola, Inc. MC68030 Enhanced 32-bit Microprocessor User's Manual. Motorola, Inc., 1987.

[MAUR1] Maurer, W. D., and Lewis, T. G. "Hash Table Methods." ACM's Computing Surveys, Vol. 7, No. 1, Mar. 1975: 5-19.

[MAUR2] Maurer, W. D. "An Improved Hash Code for Scatter Storage." CACM, Vol. 2, No. 1, Jan. 1968: 35-8.

[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 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]

```
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 char;

function Mapping_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 5. Shin's Mapping Hash Algorithm

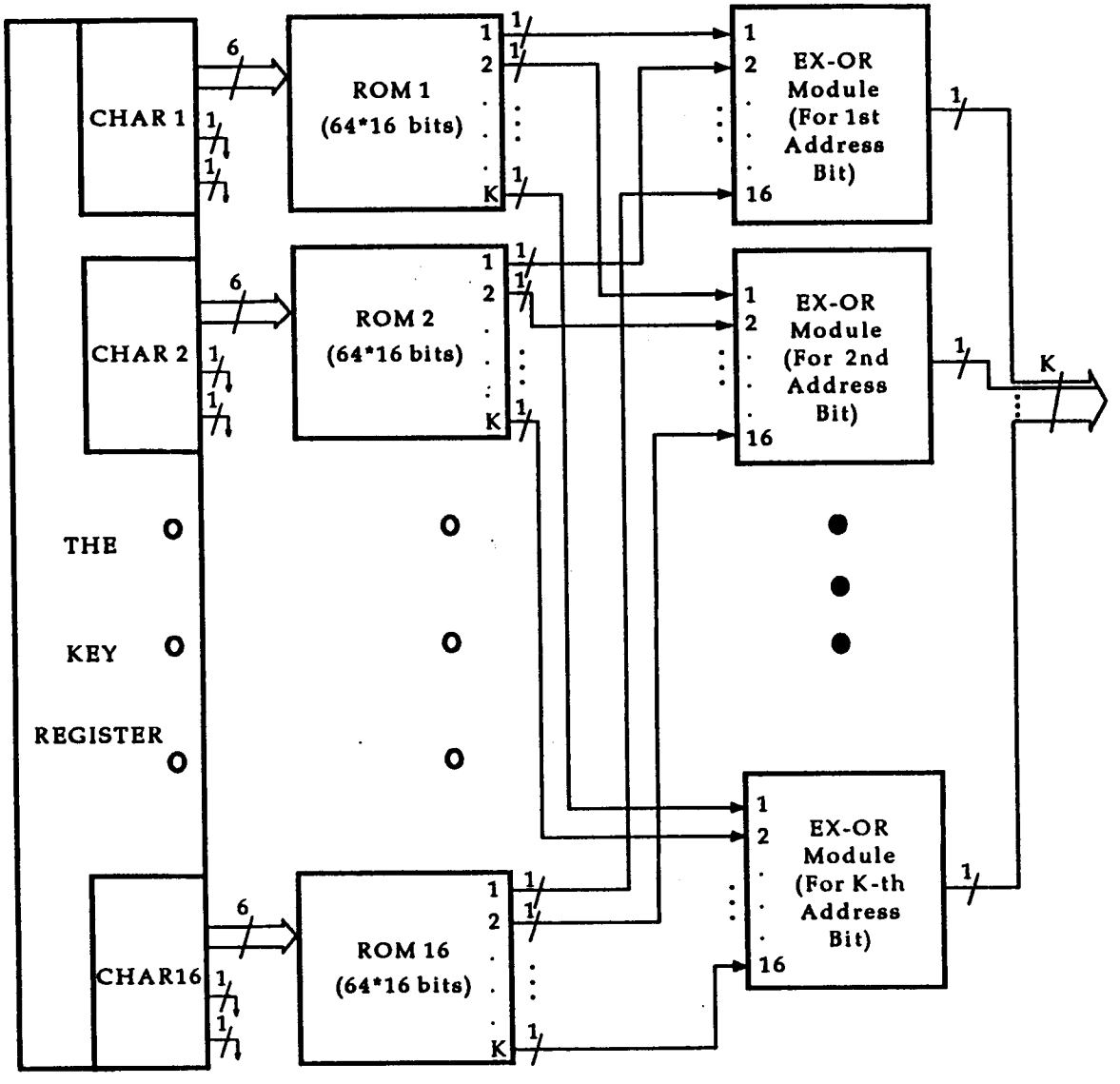


Figure 6. The Hardware Mapping Hash Coder

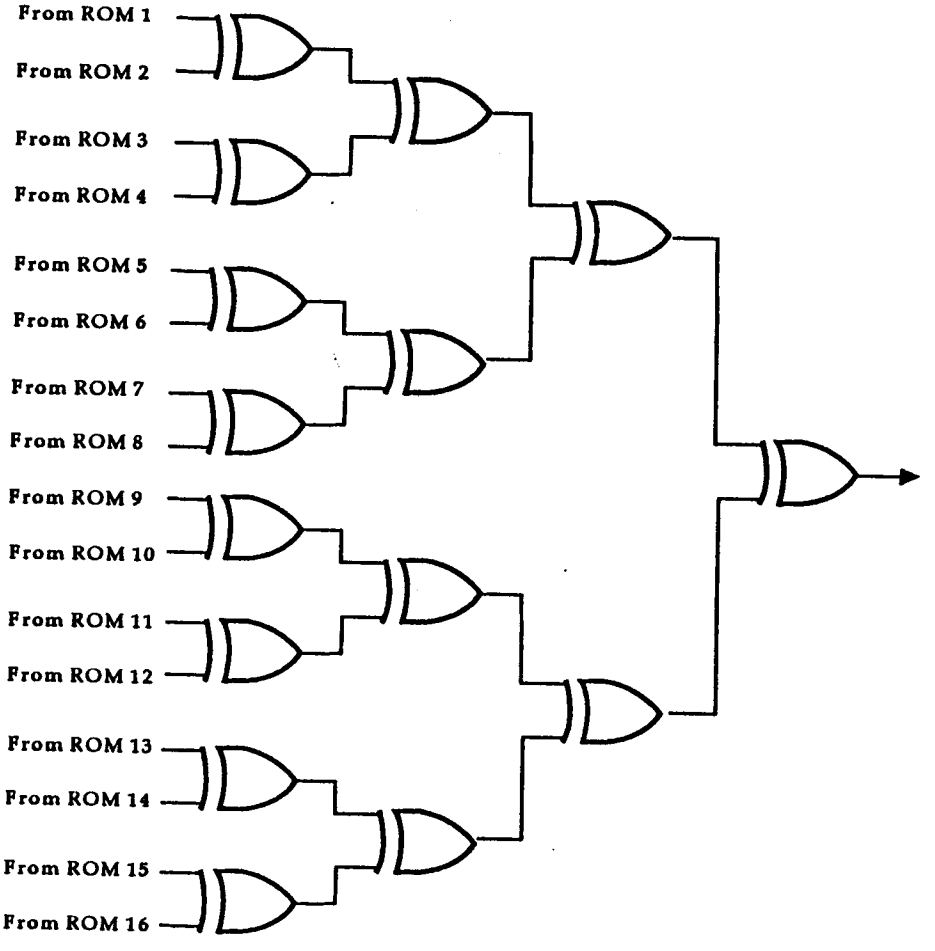


Figure 7. Exclusive-OR Module for a Hash Address Bit

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 FS(0,10,20,30) FS(0,11,22,25)	 4.20 4.03	 3.96 4.46	 4.27 4.88	 44	 1 (4)	 192
Division (Divisor = 241)	5.51	5.35	4.48	70	46 16 (5)	390 3360 (5)
Pearson's Table Indexing	20.63	21.23	21.15	82	82	280
Digit Analysis (2 and 4 bytes)	4.32 3.80	4.07 4.70	3.84 19.74	40 (6)	2 (6)	112 96
Fold-boundary	4.09	3.89	53.02	56	1 (4)	117
Midsquare	4.25	4.84	88.91	72	30 8 (7)	572 2796 (7)
Multiplicative	4.42	3.29	12.49	407	64 17 (7)	422 2892 (7)
Radix	3.97	4.05	12.36	650	390 285 (7) 120 (8)	550 3234 (7) 6498 (8)
Random	4.25	3.63	9.79	162	80 57 (7) 26 (8)	470 3138 (7) 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

Shin's Join Algorithm and HIMOD-A Database Computer

Dong Keun Shin and Arnold Charles Meltzer

Samsung Electronics Co., Ltd.
Communication Systems R&D Center
Songpa P.O.Box 117, Seoul, Korea
E-mail: dkshin@telecom.samsung.co.kr

Department of Electrical Engineering and Computer Science
The School of Engineering and Applied Science
The George Washington University
E-mail: meltzer@seas.gwu.edu

Abstract

In this paper, Shin's join algorithm which uses a divide and conquer strategy to accelerate the join and several well known join algorithms are discussed. The paper illustrates how the Shin's join algorithm efficiently filters out all the unnecessary tuples with a maximum of five readings for each join attribute. In contrast to other join algorithms, the Shin's join algorithm allows the join attribute comparisons after nearly 100 percent of the unnecessary tuples are eliminated. The Shin's join algorithm is recommended since it has less dependency problems than the hash join algorithm has and it has an inherent characteristic of parallel processing.

The Shin's join algorithm can be executed in various ways of parallel processing. The algorithm is divided into two major processes: filtering process and merging process. The major processes can be executed in parallel. Each major process can be further divided into subprocesses which can be executed in parallel. Although the Shin's algorithm is not hardware dependent, this algorithm might be implemented in a database computer, HIMOD (Highly Modular Relational Database Computer), for an efficient parallel execution of the two major processes. In an initial design phase of the HIMOD, it is simply equipped with a general purpose processor as a host and a parallel architecture join database coprocessor as a back-end processor. The host performs merging process for the join while the join database coprocessor mainly performs filtering process for the join. The join database coprocessor is designed to be a speedy filter device which further accelerates the join.

I. INTRODUCTION

Ever since the relational data model was introduced by E. F. Codd's paper [9] in 1970, not only the naturalness of the two dimensional table structure but also the usefulness of the join relational operation has been well recognized. It is necessary to emphasize the join operation in design of database management systems. Since the join operation is a time-consuming but

frequently used operation, increasing the speed of the join has been a popular issue for more than 20 years. One of the goals of our research was to provide an optimal algorithmic solution for accelerating the time-consuming join relational database operation. In this paper, Shin's algorithm will be described as a solution for the join. Moreover, a database computer which uses the join algorithm will be illustrated to show how the join algorithm can be implemented in hardware.

The join operation concatenates a tuple of the source relation (S) with a tuple of the target relation (T) if the value(s) of the join attribute(s) in this pair of tuples satisfy a pre-specified join condition, and it produces a tuple for the resulting relation (R). The join operation can be illustrated in SQL using a SELECT-FROM-WHERE clause. When X.A is used to indicate that attribute A from relation X is meant and X.* stands for all attributes of relation X, an example of the join and the query the example are

Relation S

A	B	C
d	e	f
b	d	g
h	d	b

Relation T

D	E	F
a	d	c
d	g	a

```
SELECT S.* T.*
FROM S, T
WHERE S.B = T.E
```

JOIN S, T (S.B = T.E)

Relation R

A	B	C	D	E	F
b	d	g	a	d	c
h	d	b	a	d	c

In 1977, Blasgen and Eswaran [7] described several methods for evaluating a general query, involving project, select, and join relational database operations. They compared the methods based on which method had fewer accesses to secondary storage. In their examination of the join operation, both nested-loop and sort-merge algorithms were analyzed and discussed. Because of the work of these two authors, researchers were generally convinced that a nested-loop join algorithm performed acceptably on small or large sized relations when a suitable index existed. They concluded that a sort-merge join algorithm would be the choice when no suitable index existed. Both nested-loop and sort-merge join algorithms and their actual implementations will be discussed later in greater detail.

Based on the nested-loop and the sort-merge join methods, tuples that are not necessary to the resulting relation for the join will be included until the last moment although they are not necessary to the resulting relation. Assuming that the amount of data in the source and target relations is large, but the amount of resulting tuples are relatively small, then most of the tuples in the source and target relations are not needed in producing the output for the join. However, all of those irrelevant tuples are also brought to main memory from secondary storage via the I/O channel; as a consequence the channel becomes congested, which, in turn, creates the aforementioned I/O bottleneck. Many researchers have broached designing a database filter for the join operation to reduce the problem of channel congestion. Several database computers such as CAFS [3], SURE [31], VERSO [16], and DBC [16] have been designed based on the concept

of database filtering.

Since the cost of the main memory has been substantially reduced, hash join algorithms have been recognized for a great potential. It is now a well-known fact that the join algorithm based on hashing is more advantageous than nested-loop or sort-merge join algorithms [13]. One noticeable difference of hash-based join algorithms is that they minimize the amount of data moved during the process of executing a join algorithm. In 1988, the authors had a thought that an optimal algorithmic solution for the join had not yet been found. Join algorithms, such as nested-loop, sort-merge, and hash, require frequent join attribute comparisons which may result in more data movements. The hash join algorithms has dependency problems which will be discussed later. Shin's join algorithm uses hashing for division process, and eliminates and merges data efficiently for conquer process to lessen data movements. In Section II, the four major join methods are illustrated: the nested-loop join algorithm, the sort-merge join algorithm, the hash join algorithm, and the Shin's join algorithm. The time complexities and problems of join algorithms are discussed in Section III. Section IV describes HIMOD-a database computer that may effectively perform the Shin's join algorithm. Results of performance evaluation studies are presented in Section V. Finally, in Section VI we give concluding remarks and suggest some directions for future research.

II. FOUR MAJOR JOIN ALGORITHMS

In this section, we give a brief overview of the algorithms published until now for the join operation. We describe the approaches for the join in three separate subsections.

A. The Nested-Loop Join Algorithm

The nested-loop join method is the simplest among the three major algorithms. The two relations involved in the join operation are called the outer relation (or source relation) S and the inner relation (or target relation) T, respectively. Each tuple of the outer relation S is compared with tuples of the inner relation T over one or more join attributes. If the join condition is satisfied, a tuple of S is concatenated with a tuple of T to produce a tuple for the resulting relation R.

B. The Sort-Merge Join Algorithm

Each of the source (S) and target (T) relation is retrieved, and their tuples are sorted over one or more join attributes in subsequent phases using one of many sorting algorithms (e.g., n-way merge). After the completion of the sorting operation, the two sorted streams of tuples are merged together. During the merge operation, if a tuple of the relation S and a tuple of the relation T satisfy the join condition, they are concatenated to form a resulting tuple.

C. The Hash Join Algorithm

The join attributes of the source relation (S) are first hashed by a hash function. The

hashed values are used to address entries of a hash table called buckets. The same hash function is used for the join attributes of the target relation (T). If the join attribute of a tuple is hashed to a non-empty bucket of the hash table and one of the join attributes stored in that bucket matches with the join attribute, the equi-join condition is satisfied. The corresponding tuples of the S and T relations are concatenated to form a tuple of the resulting relation (R). The process continues until all the tuples of the target relation have been processed.

D. The Shin's Join Algorithm

In Shin's join algorithm, the source and target relations are repeatedly divided (hashed or rehashed) by a maximum of five functionally different hash coders until a group of source tuples and target tuples are found to have an identical join attribute. If a group of source tuples and target tuples are found to have an identical join attribute after a division process, the source and target tuples in the group are then merged in order to produce tuples for the resulting relation.

A stack is the essential data structure used in the SOFT (Stack Oriented Filter Technique) and the Shin's join algorithm as shown in Figure 1. Each stack item consists of a pair of two hash tables: one for source tuples and the other for target tuples. The stack pointer keeps track of the top item of the stack whenever a stack item is pushed into or popped from the top of the stack. Stacks in the Figure 1 depict changes in the stack. In the process of Shin's join, a maximum of five pairs of hash tables can be created. A source hash table includes 256 bucket pointers for the linked lists of source tuples, and a target hash table also includes 256 bucket pointers for the linked lists of target tuples. Any suitable numbers can be chosen instead of five and 256. Assuming that both input relations fit in main memory (e.g., main memory database (MMDB)), they are divided into a maximum of 256 linked lists for each relation by the first hash coder, as shown in step 1 in Figure 1. After the source and target tuples are hashed by the first hash coder, the tuples in the source subset file (Si) can possibly match with only the tuples in the target subset file (Ti). If an empty subset file exists, all tuples in the pair of the subset files would be eliminated since they have no potential to be included in the resulting relation.

As shown in step 2 in Figure 1, the join attribute values of the source tuples are hashed by the second functionally different hash coder; as a result, the source tuples are stored in addressed buckets in the source hash table. Using the same hash coder, the target tuples are hashed and stored in the target hash table. While the tuples are being divided into a maximum of 256 groups, the first produced hash address is compared with the subsequently produced hash addresses to see if the produced hash addresses are the same. If so, the source tuples and target tuples are merged. After the hashing process, four kinds of pairs of buckets (ij) will be possibly created. The pairs may appear in the following combinations:

(1) The source bucket (Sij) and the target bucket (Tij) are not empty.
(2) Sij is not empty, but Tij is empty.
(3) Tij is not empty, but Sij is empty.
(4) Sij and Tij are empty.

When one of the two buckets is empty, the tuples in the corresponding bucket are unnecessary; therefore, they are filtered out. The filtering scheme is one of the major concepts in Shin's SOFT.

The Shin's Join algorithm provides the termination condition for no further division process as an idea in the SOFT. The SOFT checks if the termination condition for no further division process is satisfied. If the produced hash addresses in a group of source and target tuples are identical, the Shin's join algorithm stops dividing a group of source and target tuples and starts merging the source and target tuples. Based on the current stack level, a maximum of five functionally different hash coders (as will be shown in parallel architecture filter unit of HIMOD) might be involved in checking the termination condition. If their logical ANDed result show that only a single hash address is produced from each involved hash coder, the group of source and target tuples can be merged without final screening.

The algorithm proceeds from the pair of buckets of address 0 to the pair of buckets of address 255 checking if both source and target buckets are not empty. When both buckets are not empty, the next bucket address is saved and the tuples in the source bucket and the corresponding target bucket will be rehashed (or divided) by the next functionally different hash coder. During the rehashing process, the algorithm compares the first produced hash address with the others. If the produced hash addresses are identical, the tuples are merged; otherwise, the tuples are divided further by another functionally different hash coder.

Steps 3, 4, and 5 in Figure 1 can be explained similarly. In step 6, no available hash coder is left and unnecessary data have been filtered; therefore, the source and target tuples are merged without being rehashed. As far as data structure is concerned, the linked list data structure is better than the array data structure for the buckets in steps 2 through 5 (in Figure 1) because in these steps most of the buckets may be empty. Therefore, by using the linked lists for the buckets, memory space can be conserved.

In order to eliminate 100 percent (i.e., greater than 99.9999999999% which is equal to 1 - 1/(256**5)) of unnecessary data, join attributes are to be hashed by a maximum of five functionally different hash coders to figure out all the produced hash addresses are the same. The multiple hash calculations can be effectively implemented using a parallel architecture as discussed in the architecture of HIMOD. Therefore, two kinds of software implementation of Shin's join algorithm is left to one's choice: a maximum of five hashings for each join attribute at a time and a single hashing in each reading of a join attribute. If one uses the second way for his software implementation, the filtering effect reaches more than 99.609375% (i.e., greater than 255/256) and final screening might be needed for a merge.

As shown in Figure 2, push and pop are the names of procedures operating in the stack. The procedure push inserts a pair of source and target hash tables onto the top of the stack and increments the stack pointer. The procedure pop deletes a pair of source and target hash tables from the top and decrements the stack pointer. The stack pointer always points to the current pair of hash tables (the item at the top of the stack). By referring to the value in the stack pointer, the function Bottom_Of_Stack can tell whether the stack pointer points to the first or lowest pair of hash tables as the current item of the stack.

In the Shin's join algorithm, there are several other frequently used procedures such as Assign_Source_And_Target, No_More_Next_Bucket_Addr, and Save_Next_Bucket_Addr. The module Assign_Source_And_Target uses the header pointers of both source and target linked lists based on the saved next bucket address of the current pair of hash tables in order that the tuples in the linked lists are processed through the filter again. Each next bucket address is incremented and saved to keep track of the subsequent bucket address. Whenever the procedure

Assign_Source_And_Target is called, another next bucket address, which has non-empty buckets for both source and target hash tables, is found and saved by the procedure termed Save_Next_Bucket_Addr. As a result, the procedure push saves the contents of the current pair of hash tables and increments the stack pointer in order that the next upper pair of hash tables becomes the current one or the top of the stack.

When pop is called, the stack pointer is decremented in order that the pair of hash tables directly under the current one becomes the current pair of hash tables. After the pop, the boolean function No_More_Next_Bucket_Addr should be called in order to see if there is any saved next bucket address for the current pair of hash tables. If there is none, the current pair of hash tables is checked to see if it is the first (or lowest) one. If so, the join process is terminated by breaking the repeat loop.

The algorithm shown in Figure 2 is an explanatory version of the main module of a simulation program for the Shin's join algorithm. One can also implement the Shin's join algorithm in a recursive routine which uses a binary tree, following the convention of leftmost-child and right-sibling tree representation. The nature of recursion in the algorithm simplifies the architectural structure of the database computer which implements the Shin's join algorithm. The nonrecursive implementation of the algorithm in Figure 2 is useful and practical because the stack has only five items. The nonrecursive algorithm can also be easily implemented in a microprogram.

III. DISCUSSION

When the number of tuples in the source relation (the smaller relation) is S, the number of tuples in the target relation (the larger relation) is T, and the number of tuples in the resulting relation is R, then the time complexity of nested-loop join algorithm is O(S*T). Since upper bound of S and T is N, O(S*T) can be represented as O(N*N). The time complexity of the sort-merge algorithm is O((S+T) log (S+T)). It can be represented as O(N log N) since the total number of the input tuples (N) is S+T. Considering only the time complexities, the sort-merge join should outperform the nested-loop join. However, as mentioned before, if a suitable index exists, a nested-loop can be a choice as well according to Blasgen's analysis [7].

For parallel join operations, Bitton and his research colleagues analyzed parallel sort-merge and parallel nested-loop join algorithms and concluded that, when the sizes of the two relations to be joined are approximately the same, the parallel sort-merge algorithm is superior to the parallel nested-loop algorithm [6]. Furthermore, they added that when one relation is larger than the other, the parallel nested-loop algorithm is faster.

To derive an asymptotic time complexity for a simple hash join algorithm, the number of buckets (B) in a hash table and the number of buckets in a divided hash range (D) are also considered in addition to S, T, and R. The time complexity of the hash join algorithm is represented as O((S+T)*(B/D) + R). In proportion to cheaper main memory cost, more memory space becomes available; consequently, the number of repetitions for hashing process (B/D) are reduced since the value of D gets larger. Therefore, the time complexity for the hash join algorithm can be simplified as O(S+T+R). Assuming that R is relatively smaller than S+T, it becomes O(S+T). Since S+T is actually the total number of the input tuples(N), the

time complexity can be represented as O(N). It is believed that the time complexity of the join operation cannot be better than O(N) since join attribute in every input tuples must be read at least once.

Considering actual performances, it is hard to rely only on asymptotic time complexity analysis to measure the speed performance because I/O time, communication overhead, and a number of accesses to the secondary storage are also needed for a more accurate analysis. However, in the future when necessary hardware including main memory is sufficiently provided, the time complexity will be more reliable.

When sufficient main memory is affordable, the hash-based join has the greatest advantage [13, 23, 24]. The performance of the hash join algorithm is largely dependent on the distribution performance of a chosen hash function. If the chosen one poorly distributes the tuples, the worst case may occur. Accordingly, the dependence on a chosen hash function is absolute and large in the performance of the hash join algorithm. In contrast to the hash join algorithm, the Shin's join algorithm uses five functionally different hash coders, so it is much less dependent on the performance of a chosen hash coder than the hash join algorithm.

The hash join algorithm usually requires a flexible size hash table and a preprocessing for hash table size calculation for each hashing process. The usage of flexible size hash table is not recommended for a hardware implementation of hash table in a database machine. If it uses a fixed size hash table, performance will be greatly

Another problem resides in frequent join attribute comparisons of the hash join algorithm. The architecture of the database computer that implements the join attribute comparison might be complex. Join attribute comparisons include the comparisons for unnecessary tuples which will not be included in a resulting relation. A join attribute comparison takes much longer time than a hashing of a join attribute takes. An effective hardware hash coder such as Shin's mapping hash coder [25] can calculate a hash address within only a few machine cycles. It is strongly suggested that join attribute comparisons should be performed after all the unnecessary tuples are eliminated.

The Shin's join algorithm requires a maximum of five readings for each join attribute to determine whether the associated tuple is necessary or not. Therefore, the time complexity of the algorithm is represented as O(5*N), so it can be simplified as O(N). Comparing the Shin's join algorithm with others, one can see that none of the currently existing join algorithms effectively takes advantage of any filtering scheme while the Shin's algorithm filters unwanted data efficiently. The Shin's join algorithm requires only fixed size hash tables which are favorable for a hardware implementation. Moreover, the performance of the Shin's join algorithm has much smaller dependency in the distribution performance of a chosen hash function than that of the hash join algorithm does. Section V will describe how the Shin's join algorithm is implemented in a newly developed back-end database computer to detect and filter unnecessary data for the join.

IV. DATABASE COMPUTER HIMOD

The parallel computer architecture that facilitates a faster join and the Shin's join algorithm, which can perform best when using the architecture, are presented in this paper. The

database computer which is discussed in this paper is named as HIMOD (Highly Modular Relational Database Computer). It uses a single back-end processor fabricated in a single chip [1] in its initial stage of development. The database back-end processor (DBCP) in HIMOD is especially dedicated to the join database operation.

A. An Overview of HIMOD Architecture

Shin's join method [25] used in HIMOD is divided into two processes: filtering process and merging process. Figure 3 depicts the interface between the host and the back-end and illustrates the parallel execution of filtering process and merging process. As shown in Figure 3, the host requests a join operation of a source relation and a target relation. Then the back-end will receive the request and perform the join of the source and target relations. Filtering unnecessary tuples, the back-end transmits the pointers to the tuple lists to the host processor whenever it finds the tuples that will be included in the resulting relation. The linked source tuple(s) and target tuple(s) are retrieved from the main memory and merged by the host processor. Therefore, parallelism is exploited in the join operation so that the filtering process and the merging process are concurrently executed by the special purpose back-end and the general purpose processor (host) respectively, as is indicated in Figure 3. The idea behind the Shin's join algorithm in the database computer HIMOD contends that the host processor handles the only tuples that are necessary to be included in a resulting relation. The host processor is not burdened with carrying many join attributes and comparing them for a match. The filtering scheme in HIMOD is accomplished by the Shin's join algorithm.

HIMOD uses a Motorola 68000 family microprocessor as the host (or the front-end as one might consider) processor. The back-end processor communicates with the host processor through a protocol, which is defined as the M68000 coprocessor interface [18]. The connection between the host processor unit (HPU) and the database coprocessor (DBCP) develops from a simple extension of the M68000 bus interface. The DBCP is connected as a coprocessor to the host processor, and a chip select signal, decoded from the host processor function codes and address bus, selects the DBCP. The host processor and the coprocessor configuration is shown in Figure 4. All communications between the HPU and the DBCP are performed with standard M68000 family bus transfers. The DBCP contains a number of coprocessor interface registers, which are addressed by the host processor in the same manner as memory.

B. Architecture of the Host Processor

Since the simple M68000 coprocessor interface incorporates the design of the database coprocessor, the M68000 family microprocessor is selected for the host processor of HIMOD. The HIMOD may use a general purpose processor (e.g., a Motorola 68000 family microprocessor) as a software back-end for the join. The software back-end dedicates itself to performing only join operations. The Shin's algorithm might be implemented on the software back-end instead. In this case, there is no hardware change or enhancement on the database coprocessor except for the interface unit. The software back-end approach might be recommended for some of operations in the database management systems.

C. Architecture of the Hardware Back-End

The new hardware back-end processor is intended primarily for use as a database coprocessor(DBCP) to the M68000 family microprocessor unit (HPU). This database coprocessor provides a high performance filter unit which is designed by Shin [25]. As shown in Figure 5, the database coprocessor is internally divided into three processing elements: the bus interface unit, the coprocessor control unit, and the filter unit. The bus interface unit communicates with the host processor, and the coprocessor control unit sends control signals to the filter unit in order to execute the intended database operation. For both the bus interface unit and the processor control unit, the DBCP uses the conventions of the MC68881 and MC68882 floating-point coprocessor chips [19].

1) Bus Interface Unit

The bus interface unit contains the coprocessor interface registers (CIRs), the CIR register select and timing control logic, and the status flags that are used to monitor the status of communications with the host processor. The CIRs are addressed by the host processor in the same manner as memory. All communications between the host processor unit and the DBCP are performed with standard M68000 family bus transfers [18]. The M68000 family coprocessor interface is implemented as a protocol of bus cycles during which the host processor reads and writes to these CIRs. A MC68000 family host processor implements this general purpose coprocessor interface protocol in both hardware and microcode.

2) Coprocessor Control Unit

The control unit of the DBCP contains the clock generator, a two-level microcoded sequencer, and the microcode ROM. The microsequencer either executes microinstructions or awaits completion of accesses that are necessary to continue executing microcode. The microsequencer sometimes controls the bus controller, which is responsible for all bus activity. The microsequencer also controls instruction execution and internal processor operations, such as setting condition codes and calculating effective addresses. The microsequencer provides the microinstruction decode logic, the instruction decode register, the instruction decode PLA, and it determines the "next microaddress" generation scheme for sequencing the microprograms. The microcode ROM contains the microinstructions, which specify the steps through which the machine sequences and which control the parallel operation of the functionally equivalent slices of the filter unit.

3) Filter Unit

One of the main tasks of the DBCP is to release the host from tedious database manipulation for the relational join by filtering tuples that do not have any potential for inclusion in the resulting relation. To this end, the DBCP sends only the potential tuples to the host processor. The filter unit of the DBCP is the heart of the coprocessor in determining unnecessary data and discarding them. As shown in Figure 6, the filter unit of the DBCP includes join attribute extractor, transmittal and retrieval subunit, condition code, stack pointer register, and five functionally different mapping hash coders with associated SAT (source and target single-bit wide random access memories) and associated HAC (hash address comparator).

The join attribute extractor receives a join attribute(s) (or a tuple dependent on given data structure) and transmits the join attribute value to the five attached hash coders. Compared to well-known hash methods (e.g., the division method), the Shin's mapping hash coder distributes keys effectively and the mapping hash coder which is much faster and cheaper. Thus the hardware implementation of the mapping hash function [25] is used in the hash coder of HIMOD. The hash addresses generated by functionally different mapping hash coders using a join attribute as an input key are independent of each other, but the address calculation times required by functionally different mapping hash coders are always the same and they are only a few machine cycles. The property of statistical independence in the mapping hash function is valuable in this application.

The SAT includes two single-bit wide RAMs. One RAM (source RAM) is for a group of source tuples, and another (target RAM) is for a group of target tuples. The single-bit wide RAM has 256 bits. Each bit in a RAM is addressed by a hash address. Each SAT is connected with a hash address register in an associated HAC. The hash address register is equipped with an increment function so that the address register will keep track of the next bucket address to be processed, and will feed it to the connected SAT. Therefore, each SAT has a built-in multiplexer to select the right address at any time as is shown in Figure 7. The controller sends signals to the control lines of the multiplexer for the right selection of an address. The controller also sends memory write signals to both source and target single-bit wide RAMs. Therefore, when the tuples in the source relation are scanned, the single-bit wide source RAM is marked based on the hash address from the hash coder.

By the same system, when the tuples in the target relation are scanned, the single-bit wide target RAM is marked instead. When the multiplexer in the SAT selects a hash address from the corresponding HAC, the hash address is used to determine if one of source and target buckets is empty. This can be done by detecting hash-addressed bits if both the source RAM and target RAM are '1'. The single-bit output from the source RAM and the target RAM are then logically ANDed. The resulting single-bit output is sent to the corresponding HAC in order to determine whether or not the tuples in the source and target buckets have to be processed. If one of the buckets is empty, then tuples in those buckets will be eliminated as is described in the SOFT of the Shin's join algorithm.

The hardware structure that enhances this filtering technique also should be explained. The architecture of the DBCP is characterized by a stack oriented structure of five fairs of SAT and HAC. If there are any SATs lower than the current SAT (which is pointed by the stack pointer), they are saved in the stack and deactivated during the filtering process. The current SAT participate in the filtering process. The contents of the current SAT are cleared first and the bits in the SAT marked as the hash addresses are then produced. When a SAT is saved in the stack, a file or a list of input tuples are divided and distributed into the addressed buckets in the hash table according to the prior level hash coder in the stack. The divided list of source tuples and the list of target tuples are passed through the filter again using the current and higher HACs if their join condition attributes are not detected as identical. Thus the source and target relations are divided repeatedly, discarding unwanted tuples, until the DBCP determines that the partitioned lists of the source and target tuples have the same join attribute. Ultimately the pointers to the partitioned lists of the source and target tuples are sent to the host processor, and a series of source tuples and a series of target tuples are retrieved and merged to produce the

resulting tuples.

To efficiently determine whether or not the scanned source tuples and target tuples have the same join attribute, a HAC is attached to each of the five hash coders. The HAC is designed so that it sends a signal to the controller to stop dividing the tuples, as is explained below. The HAC consists of a hash address register which keeps a record of the first hash address produced by the corresponding hash coder and the number of exclusive-OR gates, the OR gate, and the JK flip-flop. Each incoming hash address is compared with the first produced hash address, as illustrated in Figure 8.

In order to load the first hash address, a controller sends a signal ('1') to load the first produced hash address into the hash address register. Once the first hash address is loaded, the controller does not allow other hash addresses to be loaded into the hash address register. In each HAC, the same number of exclusive-OR gates, as the number of bits in a hash address register, are needed. The first bit of the address loaded in the hash address register and that of an incoming hash address are inserted into the first exclusive-OR gate. If both are the same, the output of the exclusive-OR gate is '0.' If they are not the same; that is, if one input bit is '1' and another is '0,' then the output is '1,' and it is passed to the OR gate. The OR gate simultaneously receives all the resulting output signals from those exclusive-OR gates. If all of the resulting bits are '0,' the output of the OR gate is '0,' indicating that both hash addresses are identical. If at least one of the resulting bits from the exclusive-OR gates is '1,' then the output of the OR gate becomes '1,' signifying that the loaded hash address in the hash address register and the incoming hash address are different. Then the output ('1') of the OR gate triggers the K input of the JK flip-flop (The output of the JK flip-flop is initially cleared to be '1' by the controller.), so the output of the JK flip-flop becomes '0.' Therefore, the five structurally identical HACs in the DBCP generate output signals at the same time.

The HAC has a second purpose. If the HAC is pushed into the stack, the hash address in the HAC is used to keep track of the next bucket address to be processed. Just before the HAC is pushed into the stack, the hash address register is cleared by the controller. The first hash address is, therefore, '0,' and the bucket zero is examined if it is empty. The inverted signal from the connected SAT tells whether or not both the source and target buckets are empty. If at least one of the buckets is empty, and if the controller allows it, the inverted signal ('1') increments the hash address register. This incrementing process is repeated until a pair of non-empty buckets is found. Before the source and target tuples in those buckets are further processed, another pair of non-empty buckets is found and the bucket address is stored in the hash address register. This bucket address is stored in the stack for later use. As a result, when the HAC is stored in the stack, the associated hash address register is used to store the next non-empty bucket address.

The hardware for hash address comparison, required to detect whether all the join attributes in a file or list are identical, merits elucidation. The purpose of this hardware is to inform the controller whether or not the input file or list should be divided further. If so, the DBCP eventually sends the pointers to the source and target tuples having the same join attribute to the host processor for concatenation. As shown in Figure 9, the five HACs are stacked. Based on the value in the stack pointer register, the 5-to-1 multiplexer selects one from the five inputs. When the stack pointer designates the first (i.e., the lowest) stack level, all the outputs from the HACs are ANDed, and the resulting output of the AND gate (D) is selected

by the multiplexer. If the stack pointer specifies the second stack level, the first SAT is saved in the stack and is not written until the controller sends a memory write signal to the SAT. The output of the first HAC is, therefore, excluded from the inputs into the AND gate (C), and outputs of the second, third, fourth, and fifth HACs are ANDed. Likewise, if the indicated stack level is the third level, the first and second SATs are saved in the stack and the multiplexer chooses the output of the AND gate (B), which receives the outputs from the third, fourth, and fifth HACs as inputs. If the indicated stack level is the fifth (the highest) level, the four lower level SATs are saved and the multiplexer selects the output directly from the fifth level HAC. The stack configurations explains the stack in the SOFT shown in Figure 1.

The single bit output from the multiplexer triggers the attached JK flip-flop if, after a whole input file or list has been scanned, all the HACs, which are equal or higher than the current stack level, indicate that only one kind of hash address has been produced from each hash coder. The output value of the JK flip-flop is then sent to the controller. The controller, based on the value from the JK flip-flop, then decides either to continue a division process or to require a conquer process. In the conquer process, the controller allows transmittal and retrieval subunit to send pointers to the lists of the necessary source and target tuples to the host processor for a merge to produce resulting tuples. The transmittal and retrieval subunit figures out the pointers (addresses in main memory) to the lists of the source and target tuples using the current stack level and a saved bucket address in a selected HAC as inputs. Then the transmittal and retrieval subunit sends the obtained pointers to the wanted tuple lists to the host for a merge. Even though the output signal indicates that no further division process is necessary, there is fewer than one chance in a trillion (1/(256**5)) that the signal will pass an unwanted key. Although the final screening is not necessary due to the 99.9999999999% filtering effect, the final screening with direct comparisons by the host processor will eliminate the spurious key, if it is present. Because this chance is extremely small, the host processor will not waste time dealing with unnecessary data and the direct comparison of join attributes is not needed.

In addition to the SOFT, the HIMOD database computer may employ the hashed address bit array stores filtering technique in CAFS [3]. The previous design of the HIMOD [25] uses the filtering technique in CAFS. In order to employ the technique in the HIMOD, a new scheme for the hash coders, so called dynamic hash coders, might be needed due to differences in architectures. In this scheme, each dynamic hash coder generates statistically independent hash address based on the current stack level information. At this point, a database computer designer may consider cost versus performance trade-offs. In this paper, the filtering technique in CAFS is not included in the design of HIMOD.

The whole filter unit is designed to support the divide and conquer strategy in performing the join relational database operation. Therefore, the filter unit should know when no further division of input is necessary. A group of HACs determines whether or not the scanned tuples have the same join attribute, and provides information to the controller concerning further division process or sending the desired tuples to the host processor.

The major operation in the filter unit is the hashing for dividing and filtering tuples. A maximum of five hash coders may participate in producing hash addresses in parallel. Both the parallel architecture of the hardware back-end DBCP for the five hash coders and the parallel architecture of each hash coder can drastically reduce the execution time of the join. Since the software back-end cannot take advantage of the speed of parallel processing, one may think

about a hardware back-end DBCP before deciding.

V. SIMULATION RESULTS: A COMPARISON WITH THE CONVENTIONAL JOIN METHODS

The simulation was performed on an IBM 4381 mainframe computer based on the initial design of HIMOD. The combined data set contains 2,048 name tuples, which is read into the system in order to create both the source relation and target relation. As each tuple is scanned, the initial letter of the last name is compared to the discriminator character variable. For example, if the discriminator is set to be "K," then the name tuples whose last name initials are from "A" to "K," are inserted into the source relation, while all others will be inserted into the target relation, e.g., "L" to "Z." For each name, the last name is used as a join conditional attribute, and the hash address is calculated using only the last name. The whole name is used to produce a hash address in the hash algorithm experiments. After creating the source and target relations, the equi-join operation is performed on those input relations in order to produce the resulting relation for the join. The source, target, and correct resulting relations are printed as proof that the algorithm logically works.

As shown in Table 1, the number of tuples brought into the processor is selected as the major measurement of overall performance, although there are other factors to be considered, such as the number of disk accesses, the number of join attribute comparisons, and I/O time. Since more data movements might create frequent disk accesses, which will in turn slow the join operation, the fewer number of tuples brought into the CPU, the shorter the response time will take for the join. In this particular case, on average, the Shin's join method takes about two hundred times (3386657/17637) less data movements than the conventional nested-loop join method. The main reason for this contrast in performance is that the Shin's join method eliminates all unnecessary data (99.9999999999%) in the filtering process while dividing the source and target relations into groups of tuples so that source tuples in a group will be matched only with those target tuples in a corresponding group. Removing unnecessary data, while hashing and dividing, helps reduce data movements drastically. The conventional nested-loop join, sort-merge join, and hash join, on the contrary, carry around every tuple (even if most of them are not wanted) all the way to the last moment before they discover it is an unnecessary tuple through direct comparison of join attributes. The direct comparison of join attributes is a complex and time-consuming operation compared with a hashing of a join attribute. The hash join algorithm is not very efficient since it has to go through direct comparison of join attributes for the tuples including unnecessary ones.

The time complexity of the Shin's join algorithm is O(N). This time complexity is actually the same with that of the hash join algorithm. However, the Shin's join algorithm may outperform the aforementioned join methods because none of them includes the concept of filtering in their algorithms and they always have to compare join attributes rigorously for a merge. The repeated direct comparisons may slow the system; however, in the Shin's join method, the direct comparison of the join attributes is only allowed after all of unwanted data are cast out. The performance of the hash join algorithm becomes poor when a chosen hash function distributes keys poorly. The hash join algorithm is not recommended due to it's heavy

dependence on the chosen hash function.

Another merit of the Shin's join algorithm is that the hash table size is fixed so that one does not need to calculate a suitable hash table size for each join operation. Usually, the performance of hash join method is largely dependent on the ratio of the selected hash table size to the number of input tuples in a smaller relation. It is certainly an overhead to calculate a hash table size based on the number of input tuples prior to each hash join operation. The Shin's join algorithm is also recommended because hash join algorithm is relatively data-size dependent while the Shin's algorithm is not.

VI. CONCLUSIONS AND FUTURE RESEARCH

A. Summary and Conclusions

The major bottleneck in relational database management systems develops from the frequently used and time-consuming join operation. Thus, it is apparent that accelerating the join operation will improve the performance of relational database management systems. In this paper, four join algorithms are mainly illustrated: the nested-loop algorithm, the sort-merge algorithm, the hash algorithm, and the Shin's algorithm. The nested-loop and sort-merge algorithms were used in many database computers during the early stages of database machine development. The hash-based join algorithms become prevalent due to the affordability of the main memory.

Comparing the Shin's join algorithm with others, one can see that none of the currently existing join algorithms effectively takes advantage of any filtering scheme while the new join algorithm filters unwanted data efficiently. Moreover, the Shin's join algorithm has an advantage in a parallel processing because parallelism is one of the important characteristics of the Shin's join algorithm. In the Shin's join algorithm, filtering and merging processes can be executed in parallel, and the process of filtering tuples in partitioned linked lists can also be divided into subprocesses to be executed in parallel.

This paper describes the Shin's join algorithm and outlines an ideal approach for filtering unnecessary tuples, and discusses the highly modular relational database computer, HIMOD, equipped with a single chip back-end processor for the join operation. The Shin's join method are divided into two major processes: the filtering process and the merging process. In the early stage of HIMOD database computer development, the filtering process is performed by the back-end processor (DBCP), and the merging process is executed by the host processor whenever it receives source and target lists of tuples from the DBCP. The parallel multiprocessing was not chosen for this study due to its complex synchronization problems and lack of cost effectiveness. However, in future research it would not be excluded from the study. HIMOD may have multiple back-ends to accelerate the filtering process or may use general purpose processors as back-ends to accelerate the merging process. In the course of this research, a single join back-end processor with specialized hardware which maximizes the filtering effect during the hashing process has been developed.

The join database coprocessor repeats the division and filtering process many times in a recursive way; therefore, nearly 100 percent of unnecessary tuples are filtered. After

repetitive division and filtering processes, the remaining tuples in the source and corresponding target list have an extremely high probability of having identical join attributes. The remaining source and target tuples are sent to the host processor and merged. All other tuples are eliminated before unnecessary comparison of their join attributes begins. This elimination of unnecessary tuples substantially reduces the number of join attribute comparisons. As a result, total data movements in performing a join are radically diminished.

The output of the join simulation program shows that the Shin's join algorithm is logically correct. Furthermore, the number of tuples passed through the processor in the simulation manifests how effectively the Shin's join method has cut down data movements. The distinguishable difference between the Shin's join algorithm and other hash-based join algorithms is that, in the Shin's join algorithm, the filtering process is combined with the hashing process. Accordingly, unnecessary data are detected and filtered while other join algorithms may carry unwanted tuples up to the last moment of join attribute comparisons.

This research has produced a new database computer using the Shin's join algorithm. The algorithm will shorten the time needed for a join, since it frequently filters unnecessary data. No time-consuming direct join attribute comparison in the Shin's algorithm merits serious consideration in choosing a join algorithm for an implementation because the currently existing join algorithms has to go through tedious direct join attribute comparisons. There is another great merit in eliminating all the unnecessary tuples with a maximum of five readings for each tuple. The database computer HIMOD can further accelerate the join because it employs the parallel execution of the filtering process and the merging process to accomplish the Shin's join method. To maximize the speed of the filtering process, the filter unit in the join database coprocessor is organized as a stack. As far as the Shin's join algorithm is concerned, the stack oriented filter unit eliminates unnecessary tuples in the most effective way.

B. Future Research

For future research, a database computer with multiple back-ends using the Shin's join algorithm would be a fruitful research topic since the algorithm has an inherent characteristic of parallel processing. This research topic will be more emphasized due to a huge demand for real time DBMS (or Main Memory DBMS) with high speed networking (e.g., ATM and fiber optic networks). As shown in step 1 and step 2 of Figure 1, a single back-end processor can detect and eliminate unnecessary tuples in only one pair of linked lists at a time. If two or more identical back-ends are provided, those linked lists are processed in parallel. Thus, if the parallel processing is developed, then the speed of the join may be increased in proportion to the number of the back-ends used. One may design a software back-end or a hardware back-end for merging process that the host processor in HIMOD performs. The number of back-ends for merging process can be multiple for some DBMS applications.

The multiple back-ends database computer would outperform the multiprocessor database computer which uses a well known join methods, such as parallel nested-loop join, parallel sort-merge join, and parallel hash join methods. None of the well known methods exploits the filtering mechanism in their parallel join algorithms and none of these methods has an inherent characteristic of parallel processing while the Shin's parallel join algorithm has both advantages. A comparative study of these parallel join methods, including Shin's join algorithm, based on

the measured response time, might provide a good direction for increasing the speed of the join.

ACKNOWLEDGMENTS

We thank Domenico Ferrari, Michael Stonebraker, Manuel Blum, Michael Carey, Arie Segev, David DeWitt, Ward Maurer, Simon Berkovich, Michael Feldman, H. Jagadish, Arun Swami, and Helen Chang for a review of this article and their feedbacks.

BIBLIOGRAPHY

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

[2] Auer, H., et al. "RDBM-A Relational Database Machine." Information Systems, Vol. 6, No. 2, 1981: 91-100.

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

[4] Babb, E. "Joined Normal Form: A Storage Encoding for Relational Databases." ACM Transactions on Database Systems, Vol. 4, No. 1. Dec. 1982: 588-614.

[5] Bancilhon, F., and Scholl, M. "Design of a Backend Processor for a Data Base Machine." Proceedings of ACM's SIGMOD 1980 International Conference on Management of Data, May 1980: 93-93g.

[6] Bitton, D., et al. "Parallel Algorithms for the Execution of Relational Database Operations." ACM Transactions on Database Systems, Vol. 8, No. 3, Sep. 1983: 324-53.

[7] Blasgen, M. W., and Eswaran, K. P. "Storage and Access in Relational Data Bases." IBM System Journal, Vol. 16, No. 4, 1977: 363-77.

[8] Boral, H., and DeWitt, D. "Processor Allocation Strategies for Multiprocessor Database Machines." ACM Transactions on Database Systems, Vol. 6, No. 2, Jun. 1981: 227-54.

[9] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." CACM, Vol. 13, No. 6, Jun. 1970: 377-87.

[10] Date, C. J. An Introduction to Database Systems. Reading: Addison-Wesley, 1981.

[11] Date, C. J. A Guide to Ingres. Reading: Addison-Wesley, 1987.

[12] DeWitt, D. J. "DIRECT-A Multiprocessor Organization for Supporting Relational Database Management System." IEEE Transactions on Computers, Vol. C-28, Jun. 1979: 395-406.

[13] DeWitt, D. J., and Gerber, R. "Multiprocessor Hash-Based Join Algorithms." Proceedings of the Eleventh International Conference on Very Large Data Bases, Stockholm, 1985: 151-64.

[14] DeWitt D. J., et al. "A Performance Analysis of the Gamma Database Machine." Proceedings of the 1988 SIGMOD Conference, Jun. 1988: 350-60.

[15] Goodman, J. R., and Sequin, C. H. "Hypertree: A Multiprocessor Interconnection Topology." IEEE Transactions on Computers, Vol. C-30, No. 12, 1981: 923-33.

[16] Hsiao, D. K. Advanced Database Machine Architecture. Englewood Cliffs: Prentice Hall, 1983.

[17] Maryanski, F. J. "Backend Database Systems." ACM's Computing Surveys, Vol. 12, No. 1, Mar. 1980: 3-25.

[18] Motorola, Inc. MC68030 Enhanced 32-bit Microprocessor User's Manual. Motorola, Inc., 1987.

[19] Motorola, Inc. MC68881/MC6882 Floating-Point Coprocessor User's Manual. Englewood Cliffs: Prentice Hall, 1987.

[20] Pang, H., Carey, M. J., and Miron, L. "Partially Preemptible Hash Joins." Proceedings of the ACM SIGMOD, May 1993: 59-68.

[21] Qadah, G. Z., and Irani, K. B. "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine." IEEE Transactions on Software Engineering, Vol. 14, No. 11, Nov. 1988: 1668-83.

[22] Richardson, J. P., et al. "Design and Evaluation of Parallel Pipelined Join Algorithms." ACM SIGMOD, Vol. 16, No. 3, Dec. 1987: 399-409.

[23] Schneider, D. A., and DeWitt, D. J. "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment." Proceedings of the 1989 ACM SIGMOD, Vol. 18, No. 2, Jun. 1989: 110-21.

[24] Shapiro, L. D. "Join Processing in Database Systems with Large Main Memories." ACM Transactions on Database Systems, Vol. 11, No. 3, Sep. 1986: 239-64.

[25] 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.

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

[27] Shultz, R. K. "Response Time Analysis of Multiprocessor Computers for Database Support." ACM Transactions on Database Systems, Vol. 9, No. 1, Mar. 1984: 100-132.

[28] Smith, D. C., and Smith, J. M. "Relational Database Machines." IEEE Computer, Vol. 12, No. 3, Mar. 1979: 18-38.

[29] Stonebraker, M. R., et al. "The Design and Implementation of INGRES." ACM Transactions on Database Systems, Vol. 1, No. 3, Sep. 1976: 189-222.

[30] Stonebraker, M. R. The INGRES Papers: Anatomy of a Relational Database System. Reading: Addison-Wesley, 1985.

[31] Su, S. Y. W. Database Computers Principles, Architectures, and Techniques. New York: McGraw-Hill, 1988.

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

[33] Valduriez, P., and Gardarin, G. "Join and Semijoin Algorithms for a Multiprocessor Database Machine." ACM Transactions on Database Systems, Vol. 9, No. 1, Mar. 1984: 133-61.

[34] Valduriez, P. "Semi-Join Algorithms for Multiprocessor Systems." Proceedings of ACM's SIGMOD 1982 International Conference on Management of Data, Jul. 1982: 225-33.

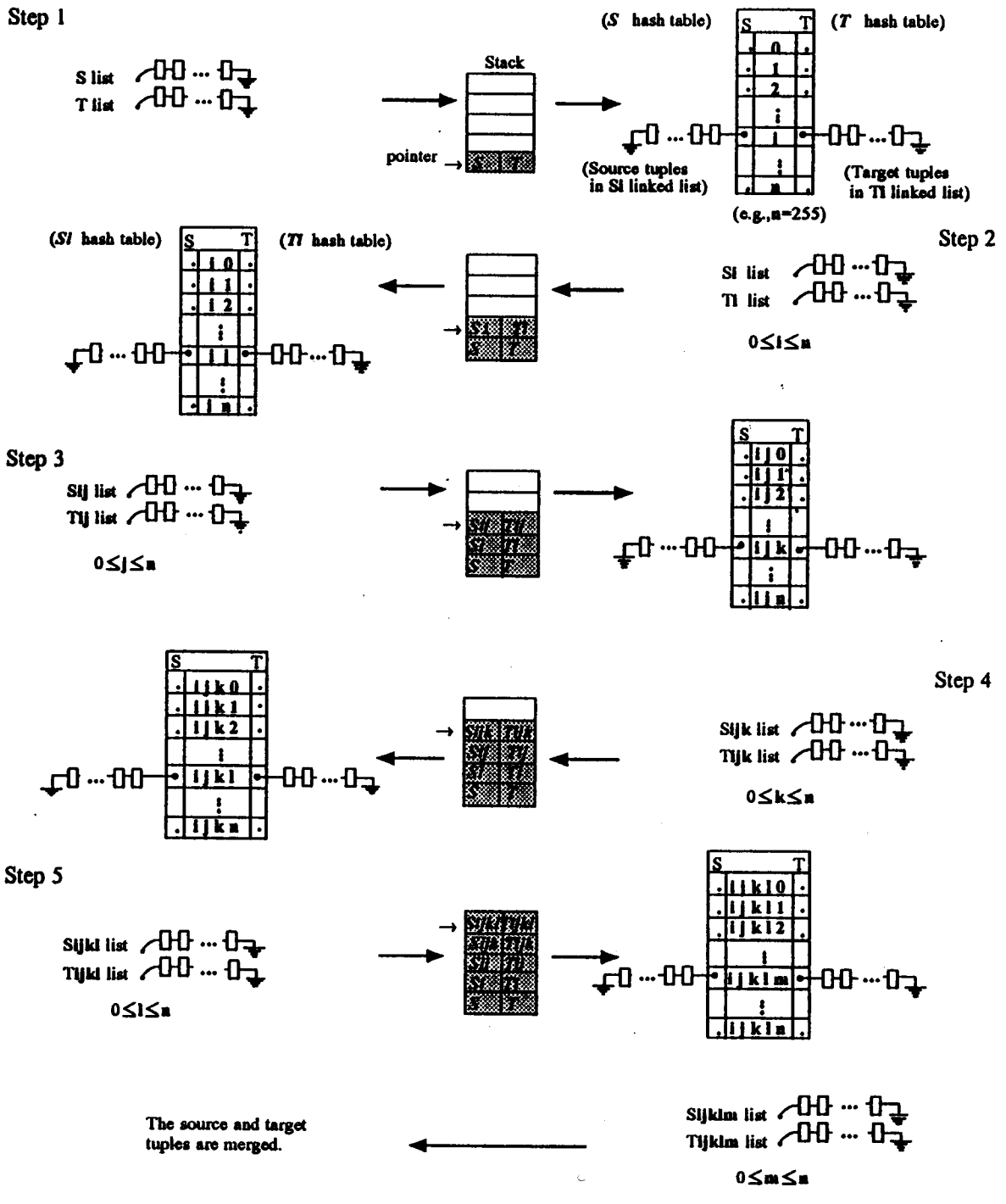


Figure 1. The SOFT and the Shin's Join Algorithm

```
begin
  Initialization;
  finish : = false;
  repeat
    Hash_Source_And_Target_Relations;
    If Only_One_Hash_Address_Produced then
    begin
      Merge_Tuples_And_Output;
      If No_More_Next_Bucket_Addr then
      begin
        If Bottom_Of_Stack then
          finish : = true
        else
        begin
          pop;
          if No_More_Next_Bucket_Addr then
          begin
            if Bottom_Of_Stack then
              finish : = true
            else
            begin
              pop;
              if No_More_Next_Bucket_Addr then
              begin
                if Bottom_Of_Stack then
                  finish : = true
                else
                begin
                  pop;
                  if No_More_Next_Bucket_Addr then
                  begin
                    if Bottom_Of_Stack then
                      finish : = true
                    else
                    begin
                      pop;
                      if No_More_Next_Bucket_Addr then
                      begin
                        if Bottom_Of_Stack then
                          finish : = true
                        else
                        begin
                          Assign_Source_And_Target;
                          Save_Next_Bucket_Addr;
                          push;
                        end;
                      end;
                    end;
                  end;
                  else
                  begin
                    Assign_Source_And_Target;
                    Save_Next_Bucket_Addr;
                    push;
                  end;
                end;
              end
              else
              begin
                Assign_Source_And_Target;
                Save_Next_Bucket_Addr;
                push;
              end;
            end;
          end
          else
          begin
            Assign_Source_And_Target;
            Save_Next_Bucket_Addr;
            push;
          end;
        end;
      end
      else
      begin
        Assign_Source_And_Target;
        Save_Next_Bucket_Addr;
        push;
      end;
    end
    else
    begin
      Assign_Source_And_Target;
      Save_Next_Bucket_Addr;
      push;
    end
  until finish;
end.
```

Figure 2. The Shin's Join Algorithm

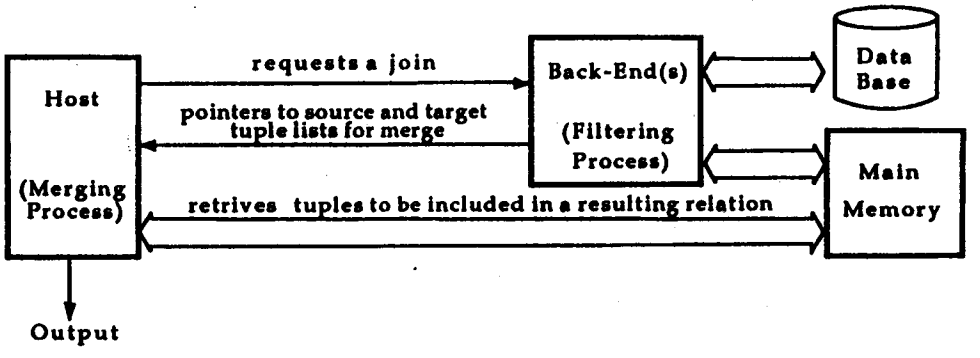


Figure 3. Execution of the Relational Join in HIMOD

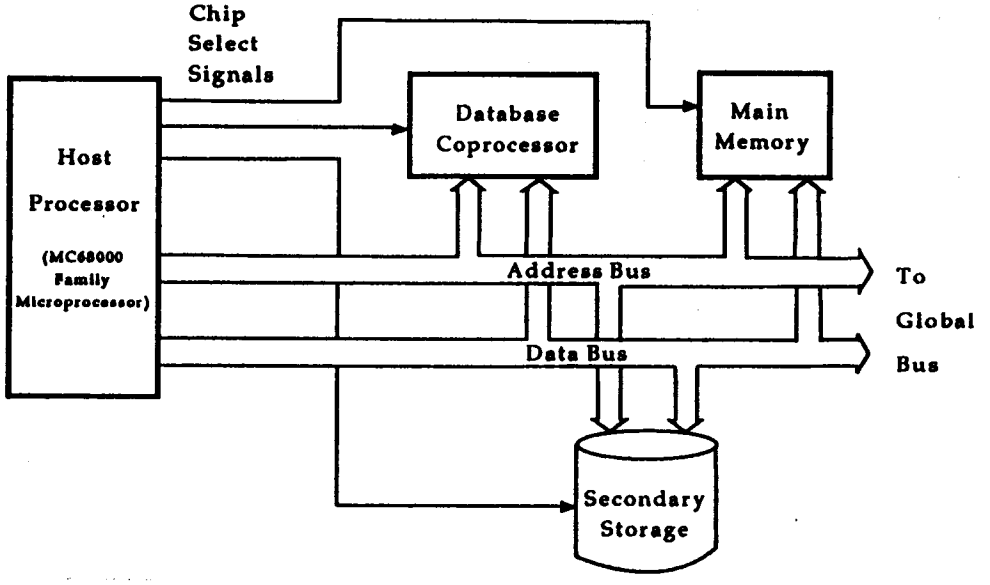


Figure 4. Coprocessor Configuration

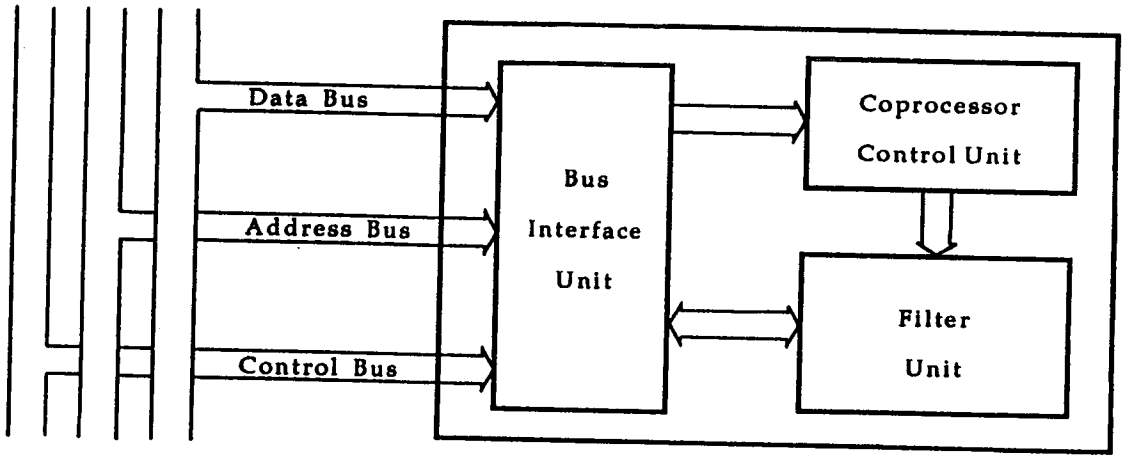


Figure 5. DBCP Simplified Block Diagram

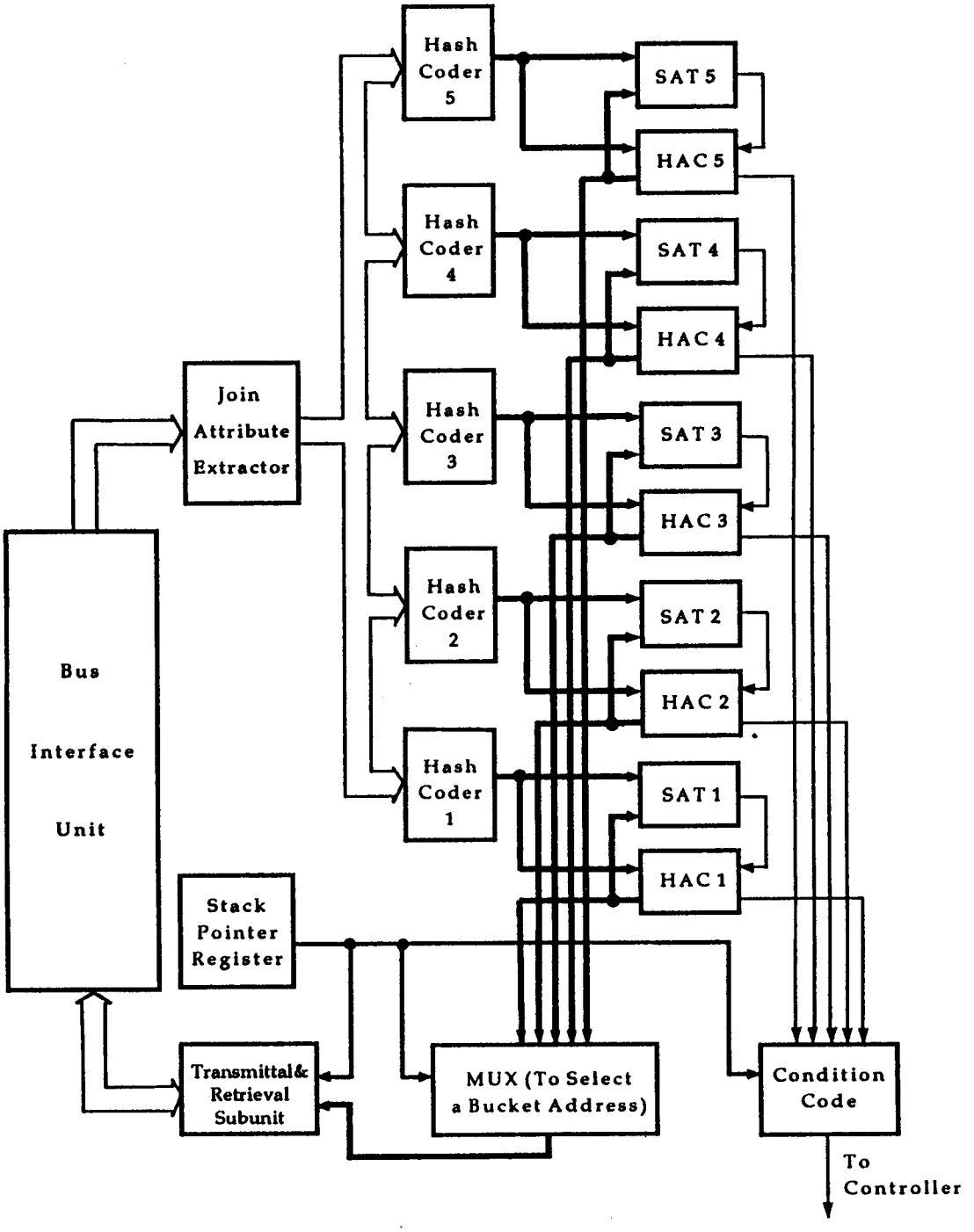


Figure 6. The DBCP Architecture (Filter Unit)

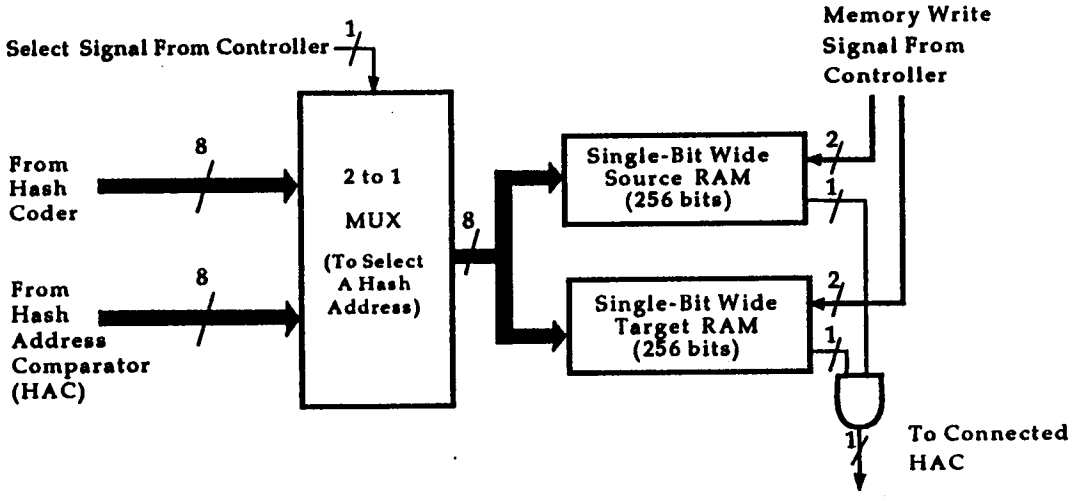


Figure 7. Source and Target Single-Bit Wide RAMs (SAT)

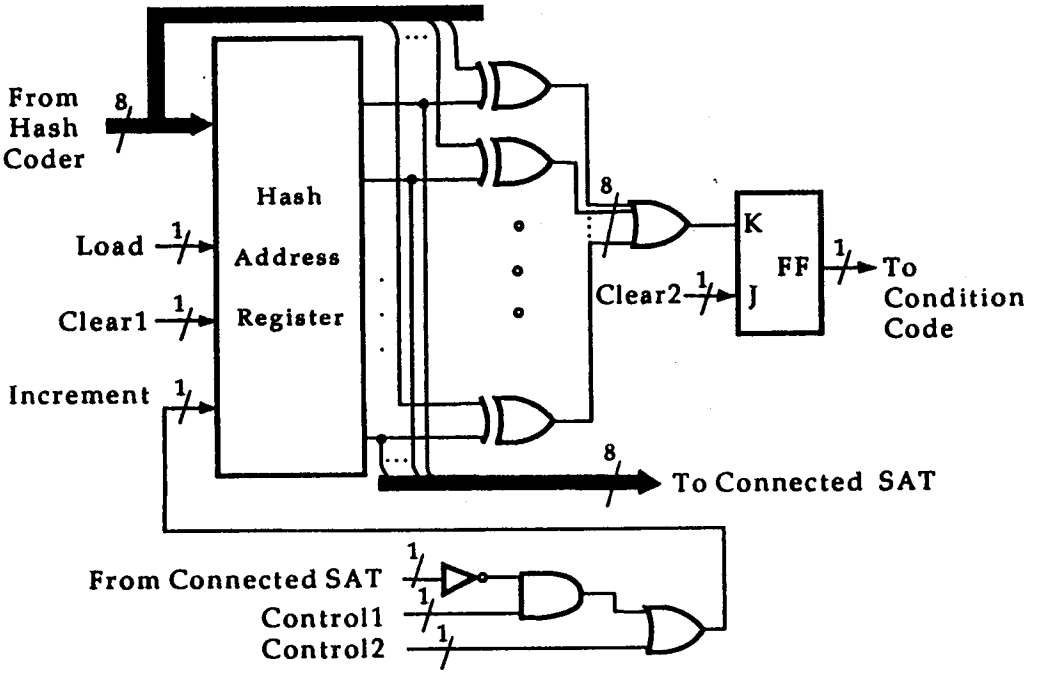


Figure 8. Hash Address Comparator (HAC)

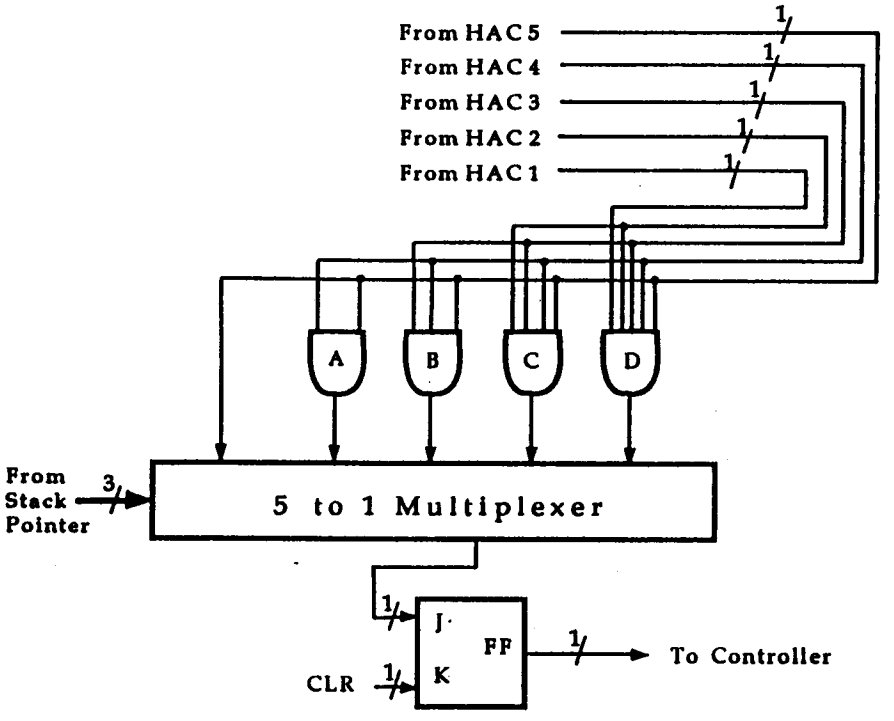


Figure 9. Condition Code for Checking if Only One Kind of Hash Address is Produced

	Number of tuples in relations			Number of tuples brought into the processor	
Discri-minator	Source	Target	Result-ing	Shin's Join in HIMOD	Nested-Loop Join
A	155	1,893	355	2,426	293,415
E	646	1,402	919	3,795	905,692
G	799	1,249	902	4,095	997,951
K	1,196	852	846	4,419	1,018,992
V	1,961	87	205	2,902	170,607
Sums:				17,637	3,386,657

Table 1. Number of Tuples Brought into the Processor