1. Original Manuscript for Dissertation

CONTENTS

Abstract.

Ever since computer have been used more for database management systems than for complex calculations, there are many efforts to provide effective database machines. Because of advantages of relational database model, most of database machines is built for that model. Among several relational database operations, the join operation is the most-time consuming and complex operation; however, it is used frequently. Many researchers have made endeavors to speed up the join, but this issue is not fully explored yet.

In this dissertation, a new hash-based join method is developed and implemented on hypothetical database machine 'Highly Modular Relational Database Computer (HIZMOD)' which is equipped with a general purpose processor as a host and a database coprocessor (DBCP) as a back end processor

The idea behind the new join method is to filter most unnecessary data as early as possible before sending them for merge. Therefore, the PBCP is designed to be a speedy filter device.

The major operation of the PBCP is the hash-based joining hashing, so a new fast hardware-oriented hash function has been developed with acceptable distribution performance to speed up the filtering process. This dissertation surveys for a hash coder of PBCP several newly developed hash functions and well known hash functions comparing them based on criteria such as speed, distribution and cost (when it is built in hardware). It concludes that the new mapping hash function is the chosen one for the hardware hash coder of the PBCP.

1. Introduction.

The purpose of this chapter is to introduce the main objectives and scope of the dissertation, to explain database machine architectures, and to describe approaches taken by existing database computer to accellerate the join relational database operation. Finally, this chapter covers the issues in designing effective hash coder for relational database back end since hashing technique becomes popular in performing join and other relational database operations.

CHAPTER 1

INTRODUCTION

1.1 Scope of Dissertation

The purpose of this dissertation is to provide the efficient and powerful method to accelerate the complex and time-consuming join relational database operation. A new join algorithm and a computer architecture which facilitate a faster join will be described. The existing join algorithms will be illustrated in chapter 2 to provide histories and alternative ways of implementing join operation. Especially, hash join methods will be discussed in depth in section 2.3 since the proposed join algorithm is also a hash based join method. Designing an effective hash coder has been demanding since hash functions are used in many database operations and other applications. Therefore, several new hashing

functions for use with a join operation are introduced and compared with current hash functions in terms of distribution, speed, and cost in chapter 3. Even though every application environment for hash coder is perhaps different, the basic approaches and principles used in implementing a hash coder in this dissertation might provide a direction or an option for the people who looking for a good hash coder. After the effort on the survey of hashing functions, Mapping hash method is selected for designing hash coder in the join coprocessor. Based on the chosen hash method, in chapter 4, a new hash based join algorithm is illustrated with the simulation results to show what are differences from and advantages over the conventional join and

other hash based join methods.

The architecture of the join hardware back end processor using the MC68030 Enhanced 32-bit Microprocessor as the front end processor will be presented in chapter 5. Also the join software back end using the MC68030 which is used for the front end processor will be explained and compared with the hardware back end in terms of speed.

Then in chapter 6, other relational operations which utilize the implemented hash coder such as project, union, intersect, and difference are discussed. Finally, summary and conclusions are provided in chapter 7.

1.2 Database Machine Architectures

1.2.1 Definition of Database Machine

The conventional approach to database management can be simply described as having both the database management system, other application programs, and operating systems run in the same host computer. The general approach of database machines (or back ends) is to off-load the database management functions from the host computer (or front end) to a directly attached special-purpose device. The database back end performs the intended database functions with specialized software and/or hardware architecture.

1.2.2 Classification of Database Machines

The back end processors can be categorized into hardware back end if there is an hardware enhancement or modification on the back end processor. On the contrary, if the database machine depends only on innovative software architectur on the back end processor which is phisically the same architecture with the front end processor, it is referred to as a software back end.

In this dissertation, the ~~(our)~~ hypothetical ~~database~~ computer named as "Highly Modular Relational Database Computer (HIMOD)" uses single back end processor which would be fabricated in a single chip.

If more than one back end is used to increase the system performance through the benefits of concurrency among the back ends, this is referred to as multiple back

end approach. Accordingly, database machines can be divided into any of single software back end, single hardware back end, multiple software back end, and multiple hardware back end based on their architectural configuration.

(2) continued on next page >

Multiple back end ——————>

(2) <New Paragraph>
Multiple back end approach has been taken by many database machines to increase the system performance by parallel processing. There are two major approaches of parallel processing in multiple back ends. The first approach is to have database management functions replicated in a number of processors, so the data are distributed to the these processors to be processed in a parallel manner. This approach is often called a multiprocessor system with replication of functions, and it is taken by the database machines such as DIRECT <DEWI1, DEWI2, BORA1>, GAMMA<DEWI4>, HYPERTREE <GOOD1>, and DBC/1012<HSIA2>. The second approach is to distribute database management functions among a number of processors so that each dedicated processor performs one or small number of functions efficiently. Those processors can be either general-

purpose processors (or software back ends) or special-purpose processors (or hardware back ends), and these functionally specialized processors speed up the intended database operations. This approach is often called a multiprocessor system with distribution of functions.

Many database computers such as RUBM <AVER1>, SABRE <VALD1, VALD2>, and DBC <HSIA1>, including the hypothetical database computer HIMOD take this approach. The database back end processor in HIMOP is especially dedicated to the join database operation which is often described as the most time-consuming operation and also frequently used operation. Since the join operation is one of the major bottleneck for the relation database management system, the database coprocessor in HIMOD is

specially designed to release this bottleneck accelerating the join. In chapter 5 of this dissertation, architectures of both hardware and software back end are discussed and compared for the effective processing for the join operation.

1.2.3. General Architecture of Back End Database Management System

The simplest form of back end database management system is shown in figure 1 to describe the connections between the host, the back end, and data base in the secondary storage. The relationship between the host and the back end is often considered as a master-slave coupling where the master is referred to as a front end and the slave is referred to as a back end. While the front end processor is busy executing application programs and operating system functions such as resource allocation, job and task management, security and integrity control, and concurrency control, the back end processor is dedicated to the time-consuming database operations and it

controls access to the database in most cases.

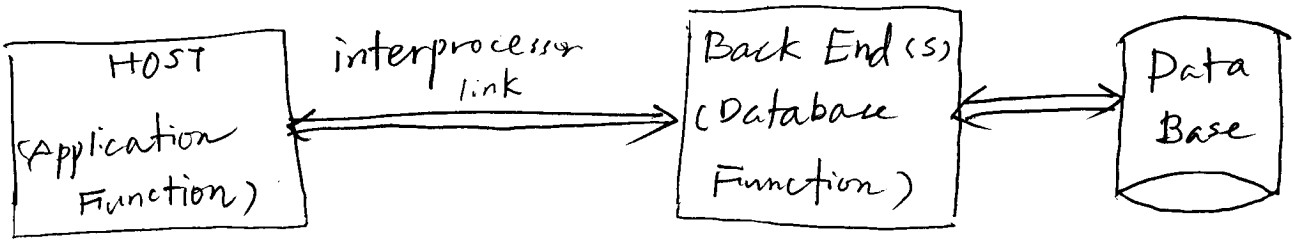


Figure 1. Back End Database Management System

The interface sequence of application programs is as follows. The application program issues a service request to the back end database management system indicating the operation required and the data to be operated upon. The back end database management system can also response to the host for a service request and receive a service from the host. If the request from the host is validated, it is served by the dedicated back end processor.

1.2.4 Other Database Machine Approaches.

1.2.4 Database Filter and Intelligent Secondary Storage Device Approaches.

Database machines can also be classified based on the kind of problem (or limitation) the group of database computers attempt to solve (or remove). Some database computers exploits parallel processing capabilities which are built into the read/write mechanisms in secondary storage devices so that data stored on these devices can be directly searched and manipulated. This approach allows the database computer to examine data locally before transferring data to the CPU. As a result, this approach can eliminate the limitations of the conventional storage devices such

that the contents of an entire relation or file in the secondary storage need to be brought to the main memory to be examined by the CPU, and can transfer only relevant data to the CPU for further processing. The database computer like CASSM, RAP, and RARES <SMIT1> use this approach by having a processing element for each track of a rotating memory device such as a disk, a drum, a charge couple device, or a magnetic bubble memory.

There is another well known bottleneck existing between secondary storage devices and main memory since huge amounts of data are transmitted via I/O channels. In addition, slow mechanical movement in secondary storage devices controls the data transfer between these devices and main memory at the same speed as the storage

devices. The database filter which resides on the I/o channel filters unwanted data from a requested file from secondary storage and sends only relevant data to the CPU. This approach lessens traffic congestion in the I/o channel. The database filter approach taken by database machine like CAFIS <BABB1>, SURE< >, and VERSO <HSIA1>.

1.3 Approaches to Speed Up the Join Relational Database Operation.

1.3.1 Complex and Time-Consuming (on the Join) ? Operation — the Join

Ever since the relational data model has been introduced by E. F. Codd's pioneering paper <CODD1>, the advantages of the relational model over the hierarchical and network models have become increasingly well recognized. Among relational operations, the join operation is very complex and time-consuming operation; however, it is frequently used. The join operation concatenates a tuple of the source relation with a tuple of the target relation if the value(s) of the join attribute(s) in this pair of tuples satisfy a pre-specified join condition.

Blasgen and Eswaran <BLAS1> described several methods for evaluating a general query involving project, select, and join relational database operations, and compared the methods based on which method had fewer accesses to secondary storage in 1977. For the join operation, nested-loop and sort-merge algorithms are analyzed and discussed. Because of their work, people were generally convinced that a nested-loop join algorithm performed acceptably on small or large sized relations when a suitable index existed, and 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 are described and discussed in details in chapter 2.

1.3.2 Filtering Technique for the Join : CAFS's Hashed Address Bit Array Stores Technique.

It is noticeable that based on both nested-loop and sort-merge join methods, unnecessary tuples which will not be included in the resulting relation for the join still be moved around until the last moment that they are no use in inclusion for resulting relation. Assuming that the data in the source and target relations are huge, but the resulting tuples are relatively small, then most of tuples in the source and target relations are useless in producing the output for the join. However, those all of those irrelevant tuples are also brought to main memory from secondary storage via the I/o channel, so the I/o channel gets congested, thereby the aforementioned I/o bottleneck is created.

Many researchers have thought about designing a database filter for the join operation to reduce the problem of channel congestion.

Among several database computers designed based on the concept of database filtering such as CAFIS <BABB1>, SURE <>, VERSO <>, and DBC <>, the hashed address bit array stores filtering technique used in CAFIS (Content Addressable File Store) filter device demonstrated dramatic improvement in all join algorithm without substantially increasing hardware cost <DEWI3, GADA2, SHAP1, VALD2, SCHN1>.

The hashed address bit array stores technique in CAFIS uses single-bit array stores which are single-bit wide random access memory. The CAFIS database machine reads the source relation, which is smaller than the target relation, and each value of the join attribute is transformed into three hash addresses by three functionally different hash coders. The three produced addresses are used to mark the corresponding

bit array store. Then the CAFIS reads the target relation which is larger than the source relation, and the three hash coders again hash each value of the join attribute. Using the three hash addresses, CAFIS verifies if the corresponding hash-addressed bits have already been set. If all three bits have been set, the join attributes of the target relation may match with those of the source relation so that those potential matched tuples are sent to the host computer to produce the tuples of the resulting relation. Otherwise, the unwanted tuples are discarded since they will not be included in the resulting relation. Among the potential matched tuples which are sent to the host processor, there could be spurious keys existing, so the final screening is left to the host processor, which must compare the two join attribute values and form the actual join by accessing the corresponding tuple pairs and concatenating them. This technique

is applied to keywords in the source and target relations only once. So if the smaller source relation is so large that most of the bits in bit array stores are set, the number of filtered target tuples would be reduced drastically. Therefore, the performance of this technique is heavily dependent on the size of source relation which must be smaller than the target relation. This data size dependency problem has been solved by the Stack Oriented Filter Technique (SOFT) which is explained in chapter 4 while the new join algorithm is being introduced.

Hash join algorithms have been recognized as having a great potential since the cost of main memory has been substantially reduced. 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 <DEWI3, BRAT1>. One noticeable difference is that hash-based join algorithms minimize the amount of data moved during the process of executing a join algorithm. Join algorithms such as nested-loop and sort-merge require frequent key comparisons which result in more data movements. The three major hash join algorithms are illustrated in section 2.3, and the time complexities are discussed in the next section 2.4.

1.4 Issues in Designing an Effective Hash Coder for Relational Database Backend.

Since the new join database coprocessor uses hash based join algorithm to perform faster join and millions of bytes might be hashed out, so designing an effective hash coder in the join database coprocessor has been another important goal of this dissertation research. Hashing has been an fruitful research area since early sixties. Major articles <MAUR1, LUM1, KNOT1> which introduce hash functions and analyze their performances are published between 1971 and 1975.

After those hashing functions are carefully examined, none of them really meets the requirements for the application for the database filter. Assuming that series of millions and millions of data are waiting to be hashed out in a database computer, any possible hardware aid can be adapted for a hash coder to calculate hash addresses as fast as possible.

One of the keys to fast calculation might be

parallel processing of each character or each bit in the whole key. This idea is feasible only if the hash coder is implemented in hardware. Fast operations such as exclusive-or operation and shift operation are recommendable, but any time-consuming serial and/or iterative computations should be avoided as much as possible to reduce address calculation time.

In hardware back end approach, the hash coder does not need to be similar to the arithmetic logic unit (ALU) of the host processor. But the cost of adapted hardware components should not be very expensive. For instance, if the cost of hash coder is more expensive than that of ALU, it does not make sense.

Of course, distribution performance should be considered in the highest priority. However, complicated and tediously long computation would not guarantee to distribute keys better than short, effective, and meaningful calculation.

To summarize the requirements for the effective database hash coder,

1. Competitive distribution performance with the current existing hash functions.
2. Extremely fast hash address calculation (e.g., few clock cycles)
3. Low cost in implementing the hash coder.

In chapter 4, various hash functions including three new hash functions are surveyed. Descriptions of hash functions are presented. And distribution, speed, and cost analysis are provided to lead the conclusion that the new mapping hash is a reasonable choice for the relational database hash coder which is implemented in hardware.

~~out hardware enhancement or modification on the backend processor, the database machine depends only on innovative software architecture on the backend processor, it is referred to as a software backend. The backend approach can choose either a software or hardware backend on either single or multiple backends.~~

~~This approach has the benefit of being cost effective; however, the cost of an additional processor must be considered. While the backend database machine increases performance through the concurrent operation of the host with backend processors and secondary storage devices, a penalty is incurred by the time of interface, communication, and data transmission of the intercomputer link. So, the concurrency must outweigh the losses caused by communication overhead. Another possible drawback is that either the host or backend processor may be overloaded while the other is idling. Therefore, the balanced resource utilization must be considered. The additional advantages of the backend database machine are reliability, due to the intermachine communication checking the facility in the backend processor, and security by the backend's authorized accessability control on data.~~

O.K.

CHAPTER 2.

2. MAJOR METHODS OF IMPLEMENTING THE JOIN OPERATION

~~2.3 Three Major Ways of Implementing the Join Relational Database Operation~~

The join operation described in section 2.1.5 has been called "explicit join" in contrast to "implicit join," which involves an explicit join followed by a Project operation over

the attributes of one of the relations. An explicit join requires a relation to be formed explicitly by concatenating attributes and values of both relations, but an implicit join can be performed by marking those tuples in the relation over which the projection is performed. So, the resulting relation is implicitly formed over the projected relation. The implicit join is similar to what has been called "semi-join" in theoretical database literature. The semi-join of of a source relation S and a target relation T on join attributes A from S and B from T is a relation R. R is formed from each tuple t c T, such that S (A) = T (B) and s c T.

This section explains three major methods of implementing the join: nested-loops, sort-merge, and hash join. Emphasis is placed on the following hash join methods: simple, GRACE, and hybrid hash.

2.3.1 Nested-Loops Join Method

This 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 inner relation (or target relation) T, respectively. Each tuple of outer relation S is compared with tuples of inner relation T over their 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.

Considering the actual implementation of the nested-loops join method, pages of tuples from both relations, instead of

only tuples existing in both relations, are read from the secondary storage and processed in order to reduce the I/O time. A page corresponds to a physical block of data such as a disk track. After the first page of outer relation S has been joined with K pages (e.g., K is a system variable: a number of tracks in a cylinder) of inner relation T, another K pages of T are read and compared with the same page of the outer relation S. The join attributes of outer relation S are compared with the join attributes of every tuple in K pages of inner relation T. Concatenations of tuples are formed in an output buffer if they satisfy the join condition. Whenever the output buffer is full, its partial outcome of the resulting relation R is written to the secondary storage. This process continues until all the pages of inner relation T have been joined with the page of outer relation S. At this time, the next page of outer relation S is read from the secondary storage, and the process of joining one page of outer relation S with all pages of inner relation T is repeated. The algorithm terminates when the last page of outer relation S has been processed.

2.3.2 Sort-Merge Join Method

Each of the source (S) and target (T) relations is retrieved from the secondary storage, and its 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 source relation S and a tuple of T satisfy the join condition, they are concatenated to form a tuple of the resulting relation R.

This sort-merge join algorithm guarantees an acceptable performance in most cases, as proven by Blasgen and Eswaran in 1977 <BLAS1>. Data statistics such as the number of tuples to be joined or the number of values in one column of a relation heavily influence the performance of the sort-merge algorithm. If the source and target relations do not fit into main memory, the performance will suffer considerably. In this case, a partial sort-merge join algorithm can be applied.

2.3.3 Hash Join Method

In this section, the general idea of the hash join algorithm is explained, followed by the issue of limited main memory size. As described in DeWitt's literature <DEWI3>, this section illustrates how the three major hash join methods such as the simple hash join, grace hash join, and hybrid hash join overcome the limitation of real memory size.

2.3.3.1 General Approach of Hash Join. In the straightforward hash join algorithm, the source and target relations, which might be named S and T, respectively, are read from the secondary storage. The join attribute values of the source relation are first hashed by a hash function. The hashed values are used to address entries of a hash table called buckets. If the

same hash function used for the join attribute value of the target relation is hashed to a non-empty bucket of the hash table, and one of the join attribute values stored in that bucket matches with the value, the equi-join condition is satisfied. The corresponding tuples of the source and target relations are concatenated to form a tuple of the resulting relation, or a pair of tuple identifications are retrieved from the bucket and are used to fetch the corresponding tuples. The process continues until all the tuples of the target relation have been processed. The accumulated tuples of the resulting relation are output to the secondary storage as the output buffer is filled.

This algorithm works best when the hash table for the source relation fits into real memory. When most of a hash table for the source relation cannot fit into real memory, this straightforward hash join algorithm can still be used in virtual memory, but it performs poorly, since many tuples cause page faults. The three hash join methods described by DeWitt <DEWI3> take into account the possibility that a hash table for the source relation will not fit into main memory.

2.3.2.2 Simple Hash Join. If a hash table containing all of source relation S fits into memory, the simple hash join algorithm is identical to the straightforward approach described above. When available real memory is not enough, the simple hash join scans S repeatedly, partitioning off as much of S as can fit in a hash table in the memory. If the join

attribute hashes into the chosen range of the hash table, insert the tuple into the addressed bucket of the hash table in main memory. Otherwise, the tuple is written to a source file on disk. Then it scans target relation T which is supposedly larger than S, and computes a hash value of each join attribute. Again, if the hash value is in the chosen range, compare the join attribute with that of S tuples in memory for a match. If the equi-join condition is satisfied, the tuples are concatenated and output. Otherwise, the T tuple is written to a target file on disk. Then it replaces relations S and T in the source and target files on the disk respectively and chooses another range, and it repeats the above steps until there are no more tuples in the target file.

2.3.2.3 GRACE Hash Join. The GRACE hash join is characterized by a complete separation of the partition phase and sorting phase. The partition phase chooses a hash function h and creates only as many buckets from source relation S as are necessary to ensure that the hash table for each bucket Si will fit into the real memory, assuming that a single block of memory is allocated as an output buffer for a bucket. Each tuples of S is scanned, hashed, and placed in the corresponding output buffer. When an output buffer is filled up, the accumulated tuples in it are written to a file referred to as subset Si in disk. After all tuples of S have been completely processed, all output buffers are flushed to disk. Then, target relation T is processed in the same way as S. If the number of buckets

equals N, the N subset files for S and the N subset files for T exit onto the disk.

During the second phase of the GRACE hash join algorithm, the actual join is performed to execute a sort-merge algorithm on each pair of subset files produced in the partition phase.

2.~~3.2~~3/.4 Hybrid Hash Join. The final type, the hybrid hash algorithm does both partitioning and hashing on the first pass over the source (S) and target(S) relations. All partitioning is completely finished in the first stage. This feature is similar to the GRACE algorithm and different from the simple hash join algorithm. However, it is different from the GRACE method in that the hybrid algorithm uses any additional available memory during the partitioning for a hash table that is processed at the same time that S and T are being partitioned, while only necessary blocks to partition S into buckets that can fit in real memory are still reserved.

The first step is to choose a hash function and set up both buffers and a hash table by allocating necessary blocks of memory. After it assigns ith output buffer block to Si for i = 1,...,n, each tuple of S is scanned and hashed with the chosen hash function. If a tuple is addressed to S0, it is placed in the hash table in the real memory. Otherwise, the tuple is moved to the Si output buffer block. Once finished, there is a hash table for S0 in main memory, and there are S1, ..., Sn files on disk.

Each tuple of target relation T is scanned and hashed

with the same hash function used for S. If the tuple is addressed to T0, the hash table is searched for a match. If there is a matched source tuple in S0, output the resulting tuple. Otherwise, the tuple is discarded. If the tuple is not addressed to S0, it belongs to Si for some i > 0, so the tuple is moved to the ith output buffer block. After this step, the subset files S1, ..., Sn and T1, ..., Tn are on disk.

Then, for i = 1 to N, the hybrid algorithm repeats the process of reading Si while creating a hash table at the same time and scanning Ti for a match to determine if the tuple is to be included in the resulting relation or to be discarded.

2.~~3.2~~3.5 Discussion. Speedwise, the hybrid hash join algorithm is the most efficient algorithm discussed here as analyzed by Shapiro <SHAP1>, when the relations are considerably large. In this algorithm, the relations are partitioned into subsets for processing in main memory. Although mechanisms exist to protect against partitioning overflow and to correct it when it happens, the hash-based algorithm requires a large main memory. However, to handle the overflow problem and reduce the size of main memory needed, the relations could be partitioned again with another hash function. Nonetheless, this rehashing is an expensive solution.

~~2.4 Hash Algorithms with Possible Hardware Aids~~

~~This section especially focuses on the algorithm which will fit well into a database hardware filter that millions of~~

2.4 Discussion

When the number of tuples in the source relation (smaller relation) is S, the number of tuples in the target relation (larger relation) is T, and the number of tuples in the resulting relation is R, the time complexity of nested-loop join algorithm is $O(S \times T)$, and the time complexity of the sort-merge join algorithm is $O(S \log S + T \log T)$ which is $O(T \log T)$ since T is greater than or equal to S. Considering only the time complexities, the sort-merge join is better. In other words, when all the tuples fit into the main memory, the sort-merge join may outperform the nested-loop join. However, as mentioned before, if there is a suitable index existed, a nested-loop can be a choice as well. based on Blasgen's analysis <BLAS17.

For parallel join operations, Bitton and his research collegues analized and concluded that when the sizes of the two relations to be joined are approximately the same, the parallel merge-sort

algorithm is superior to the parallel nested-loop algorithm. They added that when one relation is larger than the other, the parallel nested-loop algorithm is faster. Considering actual performances, it's hard to rely only on asymptotic complexity analysis to measure the speed performance because I/O time, communication overhead, and number of accesses to the secondary storage are also needed for a more accurate analysis.

To derive an asymptotic time complexity for 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 $O((S+T)\frac{B}{D} + R)$.

In proportion to more main memory space become available, the number of repetitions for hashing process (B/D) will be 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+R, it becomes $O(S+T)$. Since S+T is actually the total number of input tuples (N), the time complexity can be represented as $O(N)$.

Shapiro<SHAP1> and Schneider <SCHN1> analyzed simple, GRACE, hybrid hash join algorithms, and concluded that with respect to speed, when the relations are considerably large, the hybrid hash join algorithm is the most efficient of the algorithms discussed above. The hash-bashed join algorithm requires a large main memory. And when sufficient main memory is affordable, the hash-based join is the one that takes the biggest advantage.

CHAPTER 3

SURVEY OF HASH FUNCTIONS FOR IMPLEMENTING AN EFFECTIVE HASH CODER.

During 1970's and 1980's, implementing a powerful sorter in hardware has been an challenging topic, since an effective sorting engine can be used in many different application areas which require heavy comparative sorting processes. Whenever a specific function, algorithm, or task is repeatedly used in the extreme and it takes long process time like sorting, people often think about implementing a piece of hardware to do the job fast. since computer hardware has become much less expensive than it used to be.

In these days, distributive sorting by a hashing function is popularly used in many applications, so there have

been a demand for an effective hash coder. In this database application, an effective hash coder with an efficient hash function is essential to speed up the hash-based join operation since The hash coder will be the major component in the database filter coprocessor which series of million bytes might be passed through.

This chapter first discusses objectives in designing hash coder for use with a join operation. In the second section of this chapter, the experimental environment including encoding scheme, data sets, the measurements of distribution, speed, and cost is described. The third section introduces three new hash functions and describes their algorithms with current hash functions. The table and discussion for an analysis of distribution, speed, and cost for each hash function is provided in the fourth section of this chapter.

In conclusion for choosing a efficient hash function, the new hash function named mapping hash has been chosen for the hash coder which is implemented in hardware for relational database operations.

3.1 Objectives in Designing Hash Coder for Use With a Join Operation

The main objectives in hash function are summarized by Knuth <KNUT1>. Knuth's requirements for a good hash function include the following:

1) Its computation should be very fast.

2) It should minimize collisions.

The first requirement is crucially important in this database application since the number of join attributes the hash coder has to transform into hash addresses is assumingly very large. The hash coder is the major component of database filter that filters out irrelevant tuples transferred out of the secondary storage devices, so within the filter device, the hash address calculation per each join attribute can be a main cause of time consumption.

of minimizing collisions

The second Knuth's requirement implies that a good hash function should provide a good distribution performance. Considering the hash based join algorithms described in section 2.3, if join attributes are uniformly distributed into the buckets in the buffer, the number of accesses to the secondary storage to write tuples in a bucket to a subset file can be reduced. ① next page Since no hash function can distribute equal amount of keys in each bucket, it is necessary to compare the distribution performance of any new hash function with currently accepted hash function such as division and multiplicative hash methods. Knuth said that even though many hash methods have been suggested, none of them has proved to be superior to the simple division and multiplication methods <KNUTH>. His conclusion which is based on Lum and his colleagues' experimental results is generally accepted as true until these <LUM1>

days.

Distribution performance of some hash function might show data dependency problem. In other words, when keys are similar in any form, a data dependent hash function has a larger chance of collision occurrences. For example, by using the division method algorithm, keys like 'contract 1', 'contract 2', and 'contract 3' probably will be put into buckets next to each other. This phenomenon can be described as the data clustering due to the data dependency of division method.

To detect the data dependency, each hash method should be applied to the data set which contains many similar keys and the number of buckets should be sufficient (e.g. several hundreds).

---

① It should be noted that when dealing only with external storage, the computational efficiency of a hash function is not as important as its success at avoiding hash clashes. It is more efficient to spend microseconds computing a complex hash function at internal CPU speeds than milliseconds or longer accessing additional buckets at I/O speeds when a bucket overflows.

When a hash coder is implemented in software, requirements for a hash coder are the same with those for a good hash function as discussed above. Whereas 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 hardware hash coder might be the speed performance. This advantage, however, is largely dependent on the kind of hash function. Some hash function can be accelerated by means of hardware aids, but some gains relatively little speed while costs much more. Since hardware hash coder will be used more and more in many application areas in the future, the issue is to find out which hash function fits well into hardware implementation in consideration of speed and cost while providing a relatively good and data independent distribution performance.

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

1. Extremely fast hash address calculation.
2. Relatively good distribution performance.
3. Data independent distribution
4. Low cost in implementation.

For fast address calculation, unnecessary serial encoding scheme has to be eliminated since the process of encoding a key to another short form slows the computation. When a key is long and encoding is unavoidable, hardware encoding scheme such as arrays of exclusive-or gates which looks like downward binary tree can be used. By some circuitry attached to key registers (or join attribute registers) which contain the whole ASCII characters for a key,

the information in the bits of the key registers quickly manipulated or hashed up to produce a k bits of a hash address (when the number of buckets in a hash table is 2^k). If the circuitry is a sequential network which requires tedious serial computations, the first requirement cannot be satisfied. If the sequential network for a hash function which has a series of operation is replaced by a combinational network, the cost of the hardware hash coder will be increased exceedingly. If the algorithm of a new hash function can be processed using only a combinational circuit, it can generate a hash address within few clock cycles.

Among the current hash functions, folding and digit analysis are the hash methods that inherently fit well into combinational network, others

require sequential circuits in their nature. However, Lum said that both folding and digit analysis are erratic <LUM1>. Maurer <MAUR1> and Knott <KNOT1> suggested that the folding method should be combined with shift operations to improve the distribution performance, So this fold-shifting hash method will take more time than folding only. Digit analysis hash method is in different category such that specific data set should be analyzed beforehand to select digits, so further discussion is not necessary at this moment. Maurer analyzed that fold-shifting is probably the fastest, followed by division. The principle behind all these fast hash techniques is that folding using exclusive-or, shift, and negation using inverter are very fast and useful operations in generating hash addresses.

In section 3.3, three new hash methods are introduced. Mapping hash method is the one that I provide, and two other hash methods are given by Professor Maurer and Berkovick at the George Washington University respectively. Their algorithms and characteristics are described in the same section. This dissertation shows how mapping hash method satisfies all the requirements of a good hash coder, illustrated above.

3.2 Experimental Environment

In this experiment, it is assumed that records (or tuples) with keys (or join attributes) are moved from a sequential file in the secondary storage to a buffer memory (e.g., Data Cache) the keys are hashed by a chosen hash function, and each record is stored into a corresponding bucket in a hash table in main memory. When the bucket is filled, the accumulated records are written to the corresponding subset file. In these circumstances, the speed performance is an important criteria in comparisons of various hash methods since in this environment look up time (the time to look up a record on either the main memory or the disk) is not relatively slow compared to hash calculation time.

Three kinds of data sets are used to compare the performance of hash functions. Keys in these three data sets consists of 16 identifiable characters, and they are left justified and spave character filled.

In this experiment for a survey of hash functions, a key consists of 16 ASCII characters which is acceptable size for the keys used in most of the database applications. The first data set includes 1024 generally (or arbitrally) chosen

persons' names with 16 characters. In this data set, there are dozens of groups of people having the same last name. The second data set contains 1024 persons' names which are randomly chosen from the phone book, depending on the row, column, and page number, generated by a pseudo-random number generating function. The third data set has 1024 numbers with 16 numeric characters, which are generated by a pseudo-random number generating function. In this last data set, every key consists of only numeric characters such as '0', '1', ..., '9'.

Each character in data sets is internally represented by the corresponding ASCII code. Although ASCII code uses 7 bits, most of the computers have 8 bits for a byte or a character. So the leftmost bit always has zero value in ASCII code. In this experiment for a survey of hash functions, a key consists of 16 ASCII characters which is acceptable size for the keys used in most of the database applications.

If is assumed that 16 characters in ASCII code are initally stored in 4 words (or 16 bytes) key register. If this ASCII code character string is cosidered as a number, it must be too large for some hash functions to calculate with. Therefore, an encoding scheme is needed for those hash functions that need if. There are many encoding schemes that one can choose to use with a hash function. If a key is encoded into one word, most of the existing hashing function can be directly applied. As suggested by Maurer <MAUR1>, if keys are longer than one computer word, each word in a key can be folded to the next consecutively, taking the exclusive-OR. In other words, the highest bit of the first word of the key is exclusive-ORed with the highest bit of the second word of the key. The resulting highest bit is again exclusive-ORed with the highest bit of the third word of the key. Then the resulting highest bit is exclusive-ORed with the highest bit of

the fourth (the last) word of the key to produce the highest bit of the encoded word. The same process is applied to all other bits parallely because exclusive-OR operation is taken word by word.

This encoding sheme is advantageous over others considering that it is fast and it can be easily implemented in both hardware and software.

The number of buckets in the hash table is 256 which is 2 to the power of 8. Since this dissertation focuses on fast hardware oriented hash function, calculated hash addresses should be represented with the values of the address bits (i.e., 8 bits in this case) for some hash functions. Considering that a 16 characters key is composed of 128 bits (i.e. 16 x 8 bits) which are input to a hash function to produce 8 bits for a hash address as an output, the encoding scheme using folding with

exclusive-OR operations can be efficiently and easily implemented. It is also convincing that the choice of 256 for the number of buckets in a hash table provides a fairness for both hardware and software oriented hash functions in comparative analysis of their performances.

The purpose of this survey of hash functions is to provide the performance evaluation of the current and new hash methods clearly and concisely so that based on the results in this survey, one can make a right decision in selecting a hash function for his application. As mentioned in the previous section, the performance of a hash function can be expressed in terms of distribution, speed, and cost (when implemented in hardware). As the barometer of distribution performance, mean square deviation is selected, so each hash method is executed on

the three data sets to produce mean square deviations to show it's distribution performance. The smaller the mean square deviation, the better the distribution and the less the collisions that occur. Since the number of buckets in the hash table is 256 (2**8), if 1024 keys are hashed in uniform distribution, each bucket will contain 4 tuples, the mean. The formula of the mean square deviation is :

$$\left(\sum_{i=0}^{X-1} (N_i - M) **2 \right) / X$$

N_i : the number of tuples in bucket i

X : the number of buckets (e.g., 256 (2**8))

M : mean value (e.g., $\left(\sum_{i=0}^{X-1} N_i / X = 1024/256 = 4 \right)$

The executional speed of most hash functions can be accelerated when the hash coder is implemented in hardware for a faster computation. So two speed performances are measured in the cases of both software and hardware implementations.

When a hash function is implemented in software, execution time in clock cycles can be calculated by hands. The overall execution time for an instruction may depend on the overlap of the previous and following instructions. Therefore, in order to calculate an estimate of instruction execution time, the entire code sequence of a hash algorithm for speed evaluation is analyzed as a whole. As mentioned above the host processor of the HIMOD database computer uses a MC68030 microprocessor, the actual instruction-cache case execution times for an instruction sequence of a hash algorithm can be derived using the

instruction-cache case times listed in the tables of the MC68030 User's Manual <MOTO1>. The instruction-cache case time for most instructions is composed of the instruction-cache case time for the effective address calculation (CCea) overlapped with the with instruction-cache case time for the operation (CCop). The overlap time should be subtracted for the entire sequence as shown in the formula <MOTO1>.

later

The above formula is used in every calculation of the whole execution time that is taken to produce a hash address.

Second, when a hash function is implemented in hardware, the execution time in clock cycle is calculated for each hash function based on the following information.

The hardware hash coder may simply consists of a number of gates, and the average gate (transition) delay time for a signal to propagate from input to output through a gate is referred to as propagation (or gate) delay. On the MC68030, the gate delay time is specified as maximum 9 nanoseconds in Motorola's HCMOS (High-density complementary metal oxide semiconductor) technology, and with 20.0 MHz clock frequency for the processor speed, a clock pulse width becomes 50 nanoseconds (MC68030 processor's speed is beyond 20 MHz.). Therefore, if a circuit component of a

hash coder has the number of gate levels (L) less than or equal to 5. (i.e., $L \leq$ clock pulse width / gate delay $= 50/9$), the signals can travel from the inputs of the circuit to it's outputs passing through the series of gates within a clock cycle.

Third, the cost of a hardware impkmented hash coder is calculated simply 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. If any other device or local memory is used, it is specified in addition to the number of gates using a postfix mark.

Some hash functions use time-consuming multiplication and division operation. There have been suggestions for a fast multiplier and divider.
modular array <WALL1, CAVA1>
A fast multiplier by means of nonadditive multiply modules (NMMs) and bit slice adders, known as Wallace trees can save time in multiplication comparing

with ordinary sequential add-shift multiplier consisting of registers, a shift register, and an adder. A carry lookahead array divider also substantially increase the speed of division operation in comparison with the speed of sequential shift-subtract/add restoring/nonrestoring divider. Hardware organizations of above multipliers and dividers are explicitly explained in the referenced articles and book <WALL1, CAPP1, STEF1, CAVA1>. These fast multipliers and dividers, however, are quite expensive, so considering the speed vs. cost trade-offs, the judgement in adaption has to be made thoughtfully. For that reason, the gates of these options are also reflected in the costs of a hash coder to help a hash coder designer make a right decision.

In 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 (the ratio of the number of records to the number of record slots which are units of storage space that can hold one record) and bucket size (the number of records that can be accommodated in a location calculated from a transformation) because of the specified database application environment as explained in the begining of this section.

insertions and deletions, and

3.3 Description of New and Current Hash Functions

3.3.1 The Mapping Hash Method

The mapping hash function is a new hardware oriented hash method that converts (or maps) the internal representation (e.g., ASCII code) of each character in a key to an arbitrally chosen prime number parallely and folds these prime numbers using arrays of exclusive-OR gates to produce a number in binary form again in paralled manner. Then it extracts K bits from the binary number to produce a hash address for the hash table of 2**K buckets.

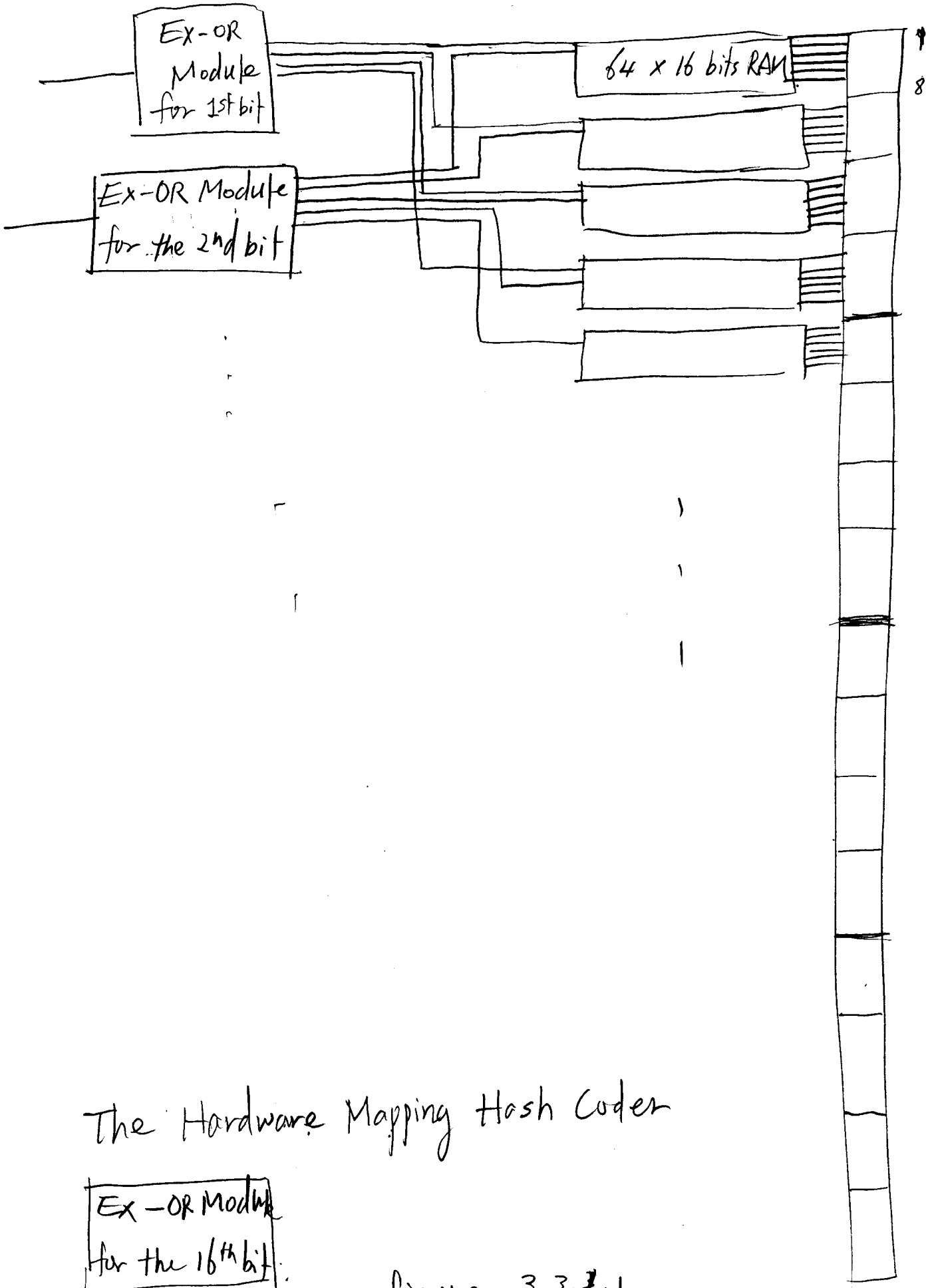
In this hash method, the nature of prime numbers which is inherent irregularity is used to randomize each ASCII character. For instance, the ASCII codes of the characters 'A' and 'B' are different in only one bit out of 8 bits. This kind of similarity in ASCII code for both alphabetic and numeric characters does not help hashing up the value for a hash address.

Therefore, when every character is converted into a prime number, the similarity between ASCII codes of the characters in a key is disappeared. Moreover, when these prime numbers are compressed (or folded) into a number, this number must be sufficiently randomized. By this folding process of random numbers, two keys which are the same except one character produces two totally irrelevant hash values. In other words, the prime number for the different character in the second key affects every bits of hashed value of a key by exclusive-OR operation, so the whole bits of hash value become completely hashed up by the character.

One of the characteristics of this hash method is that if this hash function is implemented in hardware, hash address can be calculated within few machine cycle by means of parallel processing. That hash coder

requires hardware components such as sixteen 64x16 bits RAMS (one RAM for each character) and eight exclusive-OR modules (120 exclusive-OR gates in total), as shown in figures 3.3.1.1 and 3.3.1.2.

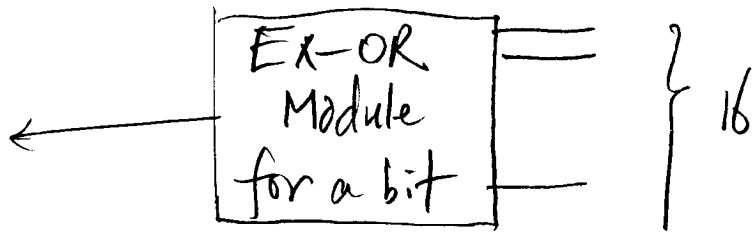
In each RAM, 64 (2^6) arbitrarily selected prime numbers are stored from the beginning. The contents of 16 RAMs can be either identical or different. Only least significant six bits of an ASCII character are used as input address to the corresponding RAM since the 7th bit is always '1' for alphabets and always '0' for numerics, and the 8th bit is not used in ASCII code. Therefore, each RAM contains only 64 words of prime numbers which are addressed by the least significant six bits of ASCII code for corresponding character in the key. When the least significant six bits of

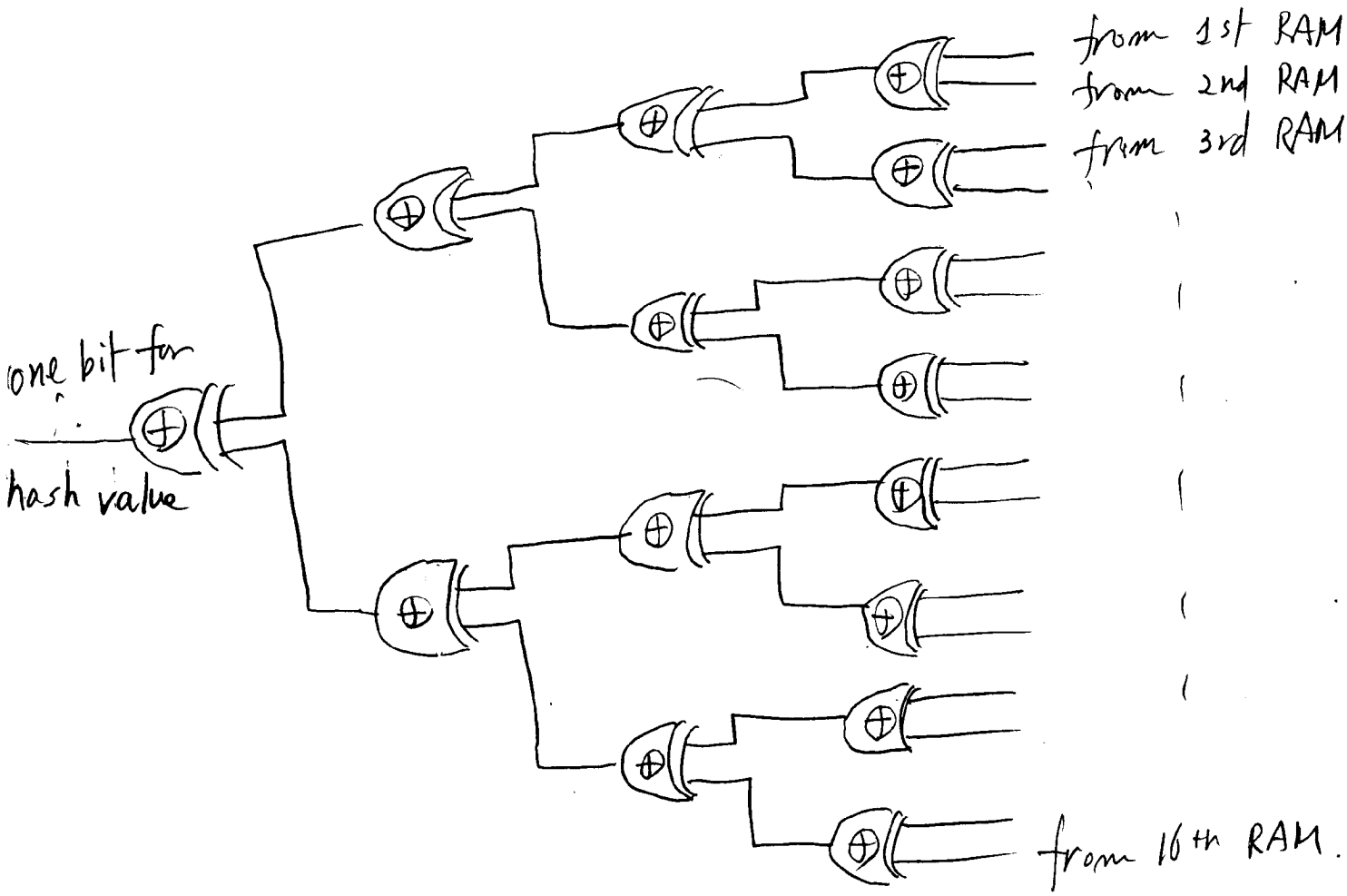


The Hardware Mapping Hash Coder

Ex-OR Module for the 16th bit.

figure 3.3.7.1
64





Exclusive-OR Module for a Hash Address Bit

figure 3.3.1.2.

each ASCII character code select a prime number in corresponding RAM, the 16 prime numbers for 16 corresponding character in a key are exclusive-ORed together to produce a final hash address parallely.

As shown in figure 3.3.1.1, 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 to generate the second bit of the resulting hash address. All other bits of a hash address are also produced at the same time.

The circuit of Ex-OR Module for each bit shown in figure 3.3.1.1 is represented in figure 3.3.1.2. To generate the first bit of the hash address, the first bit of the selected prime number from the first RAM and the first bit of the second prime number from the

second RAM are exclusive-ORed together, and the resulting bit is sent to next level of exclusive-OR. By the same way, the first bit of the third prime number and that of the fouth prime number are exclusive-ORed to produce and send a resulting bit to the next level of exclusive-OR operation. So these two resulting bits are exclusive-ORed together in the second level, and the outcome is sent to the third level to be exclusive-ORed with the resulting bit generated from the fifth, sixth, seventh, and eighth prime numbers. Then the third level resulting bit is exclusive-ORed again with the resulting bit which is produced from the eight first bits of 9th, 10th, 11th, 12th, 13th, 14th, 15th, and 16th prime numbers by the same way. Lastly, the fourth level exclusive-OR gates which receive two resulting bits from the third level

generate the final first bit resulted from the first bits of 16 prime numbers.

As shown in figure 3.3.1.1, while the first bit is being calculated, the other bits are also being computed through the four level of exclusive-OR gates parallely. The concurrent processing in looking up random numbers and in bit calculations for a hash address makes hash address computation remarkably fast.
The major operations in this hash method are indexed memory read and exclusive-OR, and they are also time-saving operation.

The conversion of an ASCII character to a random number is certainly a help randomizing the value of bits. It is, however, necessary to be cautions about the first bit of every prime number. (it is always '1'.) since prime numbers are odd number.

Thus, the resulting first bit (always "0") should be excluded in forming a hash address. One remedy for this situation is to add 1 to the prime numbers of even number addressed words in every RAMs. On that ground, the resulting first bit might be randomized enough to be included in composing bits for a hash address.

This hash method can be implemented in software using available instructions. The algorithm of this hash method in a Pascal-like notation is shown in figure 3.3.1.5.

```
const
    MAX_NO_CHARS_IN_KEY = 16 ; { number of
                                 chars in a key }

    MAX_NO_BUCKETS = 256 ; { number of buckets
                             in the hash table }
    NO_PRIMES_IN_RAM = 64 ; { number of p
type
    Key_Array_Type = array [1..MAX_NO_CHARS_IN_KEY]
                     of char;

var
    Prime_Table : array [0..NO_PRIMES_IN_RAM] of
                  integer;
```

```
procedure Mapping_Hash (Key : Key_array_Type; var Hash_Addr
                                                : integer);
var Temp, i, Index : integer;
begin
  Temp := 0;
  for i := 1 to MAX_NO_CHARS_IN_KEY do
    begin  Index := ord (Key[i]);
           if Index ≥ NO_PRIMES_IN_RAM then
                Index := Index - NO_PRIMES_IN_RAM;
           Temp := Ex_Or (Prime_Table [Index], Temp);
    {or use " Temp := Prime_Table [Index] + Temp;"}
    end;
  Hash_Addr := Temp mod MAX_NO_BUCKETS;
end;
```

figure 3.3.1.5

Programming language PASCAL provides ORD function which converts a character to corresponding ASCII integer number. Only six least significant bits of the ASCII numbers are used as indices to the table containing 64 ($=2^6$) prime numbers. The following statement in the algorithm "Temp := Ex_Or (Prime_Table [Index], Temp);" does exactly the same work that hardware implemented mapping hash method does. This assertion will soon be proved in this section. If the Ex_Or function is explained in high level language terms, it receives two integer numbers to be exclusive-ORed, converts them to two strings of 1's and 0's, takes exclusive-OR on the bits of the same position in the two strings, and convert the resulting binary string back to integer output to be sent to the calling program.

As a matter of fact, the exclusive-OR operation is valuable in hardware implementation of the mapping hash method. If anyone wants to implement this hash function in a high level language disregarding the speed, use the following statement

" Temp := Prime_Table [Index] + Temp ; "

in place of the prerious Ex-Or statement.

Then, this hash method is referred to as additive mapping hash method which gives as good distribution performance as mapping hash method provides as shown in section 3.4..

In the last statement, the time-consuming mod operation to get a remainder after a division is not necessary if the least significant k bits from the sum or the combination can be extracted for a hash address ranging between 0 and $2^k - 1$ as in low level languages. This is because the sum or the combination

in the variable Temp is already hashed up enough.

Now, the assertion that parallel processing with exclusive-OR gates as shown in figures 3.3.1. # and #. has the same effect with serial processing with exclusive-OR operations has to be proved. In other words, taking serial exclusive-ORs on 16 prime numbers gives the same result that collect all bits in the same bit position of 16 prime numbers, take exclusive-ORs parallelx passing through the EX-OR module (shown in figure #), and produce resulting bits from EX-OR module for a hashed value. It makes it easy if one considers how the 1st resulting bits in the serial and the parallel processing cases are produced and how their results are the same. Let's say X_1 is the first bit of the first prime number, X_2 is the first bit of the second prime number, and so on, so X_i is the first bit of the i-th prime number. The assertion to be proved can be expressed with the following equation:

$$(((X_1 \oplus X_2) \oplus (X_3 \oplus X_4)) \oplus ((X_5 \oplus X_6) \oplus (X_7 \oplus X_8))) \oplus$$
$$(((X_9 \oplus X_{10}) \oplus (X_{11} \oplus X_{12})) \oplus ((X_{13} \oplus X_{14}) \oplus (X_{15} \oplus X_{16})))$$

$$= (((((X_1 \oplus X_2) \oplus X_3) \oplus X_4) \oplus X_5) \oplus X_6) \oplus X_7) \oplus X_8) \oplus X_9) \oplus X_{10}) \oplus X_{11}) \oplus$$
$$X_{12}) \oplus X_{13}) \oplus X_{14}) \oplus X_{16}$$

The left hand side of the equation represents taking parallel exclusive-ORs on the first bits from the 16 prime numbers.

And the right hand side of the equation represents taking serial exclusive-ORs on the first bits from the 16 prime numbers. ✓ The right hand side can be simplified as follows: (next page)

R.H.S =
$$X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_8 \oplus X_9 \oplus X_{10} \oplus X_{11} \oplus$$
$$X_{12} \oplus X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16}$$

So the equation to be proved becomes the following:

$$(((X_1 \oplus X_2) \oplus (X_3 \oplus X_4)) \oplus ((X_5 \oplus X_6) \oplus (X_7 \oplus X_8))) \oplus$$
$$(((X_9 \oplus X_{10}) \oplus (X_{11} \oplus X_{12})) \oplus ((X_{13} \oplus X_{14}) \oplus (X_{15} \oplus X_{16})))$$
$$= X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_8 \oplus X_9 \oplus X_{10} \oplus X_{11} \oplus$$
$$X_{12} \oplus X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16}.$$

(1)

By the associative law of exclusive-OR, such that

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z,$$

Again use the associative law of exclusive-OR in the following steps.

$$(((X_1 \oplus X_2) \oplus (X_3 \oplus X_4)) \oplus ((X_5 \oplus X_6) \oplus (X_7 \oplus X_8))) \oplus$$
$$(((X_9 \oplus X_{10}) \oplus (X_{11} \oplus X_{12})) \oplus ((X_{13} \oplus X_{14}) \oplus (X_{15} \oplus X_{16})))$$

$$= ((X_1 \oplus X_2 \oplus (X_3 \oplus X_4)) \oplus (X_5 \oplus X_6 \oplus (X_7 \oplus X_8))) \oplus$$
$$((X_9 \oplus X_{10} \oplus (X_{11} \oplus X_{12})) \oplus (X_{13} \oplus X_{14} \oplus (X_{15} \oplus X_{16})))$$

Since $(X \oplus Y) \oplus Z = X \oplus Y \oplus Z$ by the associative law.

$$= ((X_1 \oplus X_2 \oplus X_3 \oplus X_4) \oplus (X_5 \oplus X_6 \oplus X_7 \oplus X_8)) \oplus$$
$$((X_9 \oplus X_{10} \oplus X_{11} \oplus X_{12}) \oplus (X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16}))$$

Since $X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$ by the associative law.

$$= (X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus (X_5 \oplus X_6 \oplus X_7 \oplus X_8)) \oplus$$
$$(X_9 \oplus X_{10} \oplus X_{11} \oplus X_{12} \oplus (X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16}))$$ again by the associative law

$$= (X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_8) \oplus$$
$$(X_9 \oplus X_{10} \oplus X_{11} \oplus X_{12} \oplus X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16})$$ by the associative law

$$= X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_8 \oplus$$
$$(X_9 \oplus X_{10} \oplus X_{11} \oplus X_{12} \oplus X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16})$$ by the associative law

$$= X_1 \oplus X_2 \oplus X_3 \oplus X_4 \oplus X_5 \oplus X_6 \oplus X_7 \oplus X_8 \oplus$$
$$X_9 \oplus X_{10} \oplus X_{11} \oplus X_{12} \oplus X_{13} \oplus X_{14} \oplus X_{15} \oplus X_{16}$$ by the associative law

So it has been proved that the first resulting bits of parallel processing and that of serial processing has the same bit value. This proof can be applied to all other resulting bits of parallel and serial processing.

Therefore, it is proved that the hardware implemented mapping hash coder in parallel processing and the software implemented mapping hash coder in serial processing produce the same hash address if they receive an identical key since they uses functionally the same hash address calculation method.

Maurer's shift and folding Hash Method

This is also hardware oriented hash method. The three primary operations in this hash method are shift (or rotate) right, exclusive-OR, and load into register, and they are relatively fast operations. As shown in figure 3.3.1.4, a key register which can contain bit information of a whole key (i.e, Eight bits for each of 16 characters; total 128 bits), the same size register with the key register for fast shift operations, and a bunch of exclusive-OR gates (128 gates) are required in the hash coder. The algorithm is the followings:

N := N Ex-OR (ROT-R(N, 1))

N := N Ex-OR (ROT-R(N, 3))

N := N Ex-OR (ROT-R(N, 7))

N := N Ex-OR (ROT-R(N, 15))

N := N Ex-OR (ROT-R(N, 31))

N := N Ex-OR (ROT-R(N, 63))

N := N Ex-OR (ROT-R(N, 127))

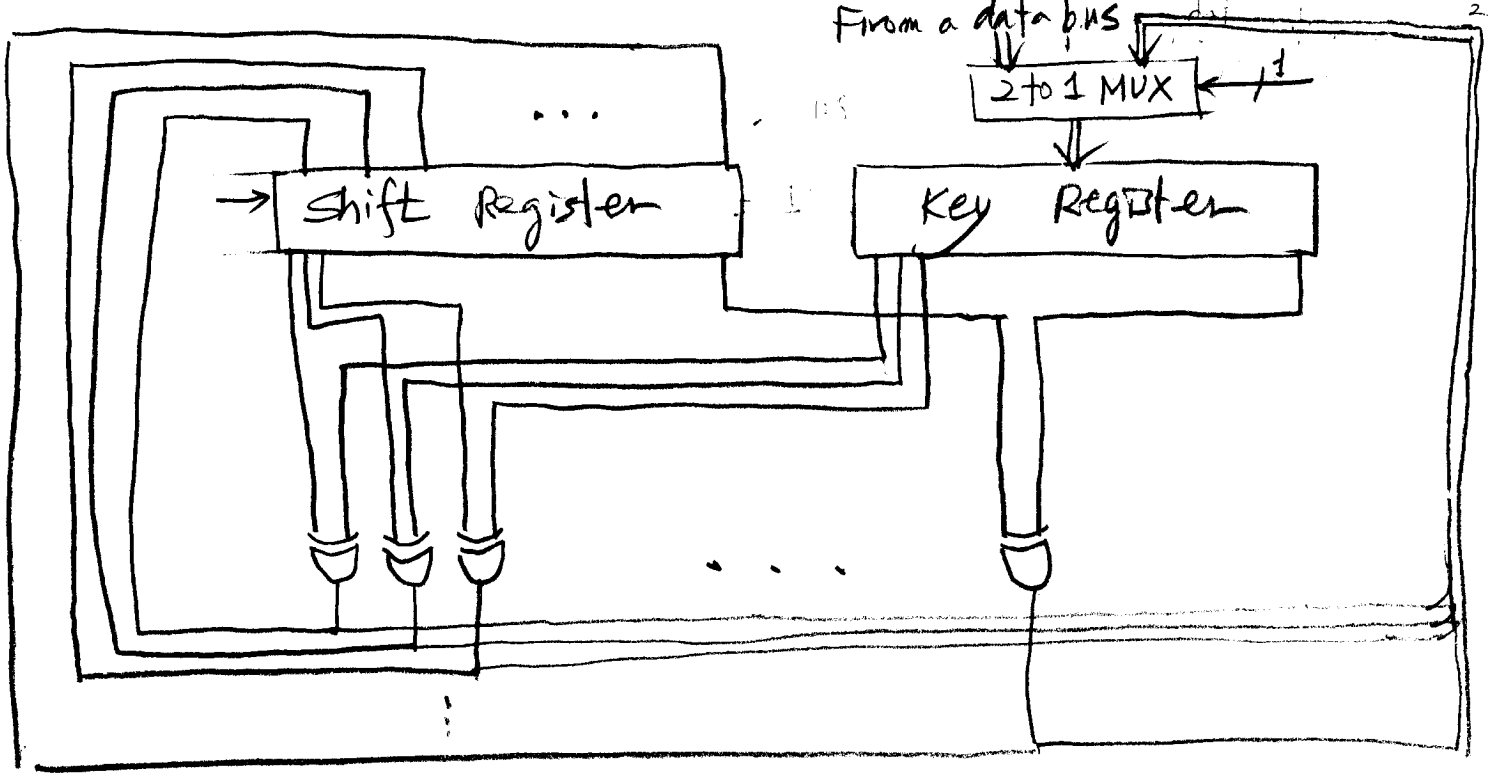


Figure 3.3.1.4

Initially an input key in both shift and key registers. The shift register will rotate the bit contents one bit right, so the right most bit will be stored in the left most bit in the shift register. Then every pair of bits in the same position of the key and the shift registers are exclusive-ORed together, and the resulting bits are loaded into both the shift and the key registers. As specified in the algorithm, in the second rotating, all the bits in the shift register are rotated 3 bits right, and exclusive-ORing and loading operations follow by the same way as above. Then rotate 7 bits right and do the same exclusive-ORing and loading. Then rotate 15 bits right and do the same, 31 bits, and 63 bits.

The numbers of bits(K_i) being rotated right each step are determined by the following method.

$K_1 = 2^1 - 1 = 1$

$K_2 = 2^2 - 1 = 3$

$K_3 = 2^3 - 1 = 7$

$K_4 = 2^4 - 1 = 15$

$K_5 = 2^5 - 1 = 31$

$K_6 = 2^6 - 1 = 63$

$K_7 = 2^7 - 1 = 127 < 128$ (N: Number of bits in the key)

If there are N bits in a key, $\log_2 N$ times of shift, exclusive-OR, and load operations are required since $K_i = 2^i - 1 < N \quad (i \geq 1)$,

$\therefore \quad 1 \leq i \leq \log_2 N$.

Berkovich's Hu-Tucker Code Hash Method

In this hash method, Hu-Tucker variable length code <KNUT2> is used. Converting each character in a key to it's 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. In this process, string size of code for each character must be added to provide the total number of bits in the final string of the code later.

This resulting string of bits is partitioned into substrings which are the same length with a hash address. The last substring might be shorter, but filled with zeros. These substrings are folded one by one taking exclusive-OR. The bits in the resulting string represent a hash address.

The idea behind this hash method is that the variable length and irregular pattern of the Hu-Tucker code for each character helps randomizing the bit values of a fixed length string for a hash address.

␣	000	n, N	1010	0	000
a, A	0010	o, O	1011	1	001
b, B	001100	p, P	110000	2	010
c, C	001101	q, Q	110001	3	011
d, D	00111	r, R	11001	4	1000
e, E	010	s, S	1101	5	1001
f, F	011100	t, T	1110	6	1010
g, G	011101	u, U	111100	7	1011
h, H	0111	v, V	111101	8	110
i, I	1000	w, W	111110	9	111
j, J	10001000	x, X	11111100		
k, K	10010001	y, Y	11111101		
l, L	100101	z, Z	11111111		
m, M	10011				

Hu-Tucker Codes.

Figure 3.3.3.#

The Algebraic Coding Hash Method

Each digit of a key is regarded as a polynomial coefficient. If a key has n digits (or bits) long, the degree of the polynomial becomes $n-1$. For example, the key 247935 is considered as $2x^5+4x^4+7x^3+9x^2+3x+5$ which is the polynomial of degree 5 ($n=6$).

The polynomial is divided by another constant polynomial of degree $m-1$ ($m \leq n$). The coefficients of the remainder which is the polynomial of degree p. ($p \leq m-1$) form a hash address.

This method of key transformation uses ideas in the theory of error-correcting codes. Hanan and Palermo <HANA> applied the theory of Bose-Chaudhuri codes to a hashing technique. In their method, the key and addresses are represented as polynomials

$$K(x) = \sum_{i=1}^{n} a_i x^{i-1}, \qquad R(x) = \sum_{i=1}^{n} p_i x^{i-1}$$

Let α be a primitive element of GF(28) (Galois Field) and consider the polynomial

$$g(x) = (x-\alpha)(x-\alpha^2)\cdots(x-\alpha^{d-1}) = \sum_{i=0}^{d-1} g_i x^i, \text{ where } d \le D.$$

(distance)

The Bose-Chaudhuri theorem <HANA1> says that $R(x)$ is the remainder of the division of $K(x)$ by $g(x)$, that is,

$K(x) = Q(x)g(x) + R(x)$, degree of $R(x) < d-1$,

then the minimum distance between two keys giving the same R is at least d. In this way, all keys which are distance D or less apart have distinct remainders, that is, distinct hash addresses.

To implement this method, the division $K(x)$ by $g(x)$ can be performed by a computer or by an α stage shift register.

The Digit Analysis Hash Method

This hash method is different from all others in such that it deals only with a static file where all the keys in an input file are known beforehand. Using either mean square deviation or standard deviation, the skewed distribution of each digit (or position) can be analyzed. Digits which have the most skewed distributions (larger deviations) are deleted to make the number of digits left small enough to produce an address in the range of the hash table. This statistical analysis does not guarantee the uniform distribution, but provide a better chance of giving a uniform spread.

Division Hash Method

This is currently the most famous and frequently used hash method. As far as distribution performance concerned, people believe that no hash function is superior to the division method because most of the researchers in this field said so <BUCH1, LUM1, KNUT1, MAUR1>.

This hash algorithm simply adds up (or exclusive-ORs) the ordinal number of words in a key and takes the remainder, dividing the sum (the combination or the encoded key) (K) by bucket size number B. So, the resulting remainder (h(K)) could represent any bucket number 0 through M-1.

Buchholz suggested that the divisor should be the largest prime number smaller than B <BUCH1>. Lum and his research colleagues say that the

divisor does not have to be a prime; a nonprime number with no prime factors less than 20 will work as well <LUM 17.

The Folding Method

In this hash method, the key is partitioned into several parts (e.g., 3), partitions in the key are folded inward like folding paper, as shown in figure 3.1.1.2. Bits (or digits) falling into the same position are exclusive-ORed (or added) together.

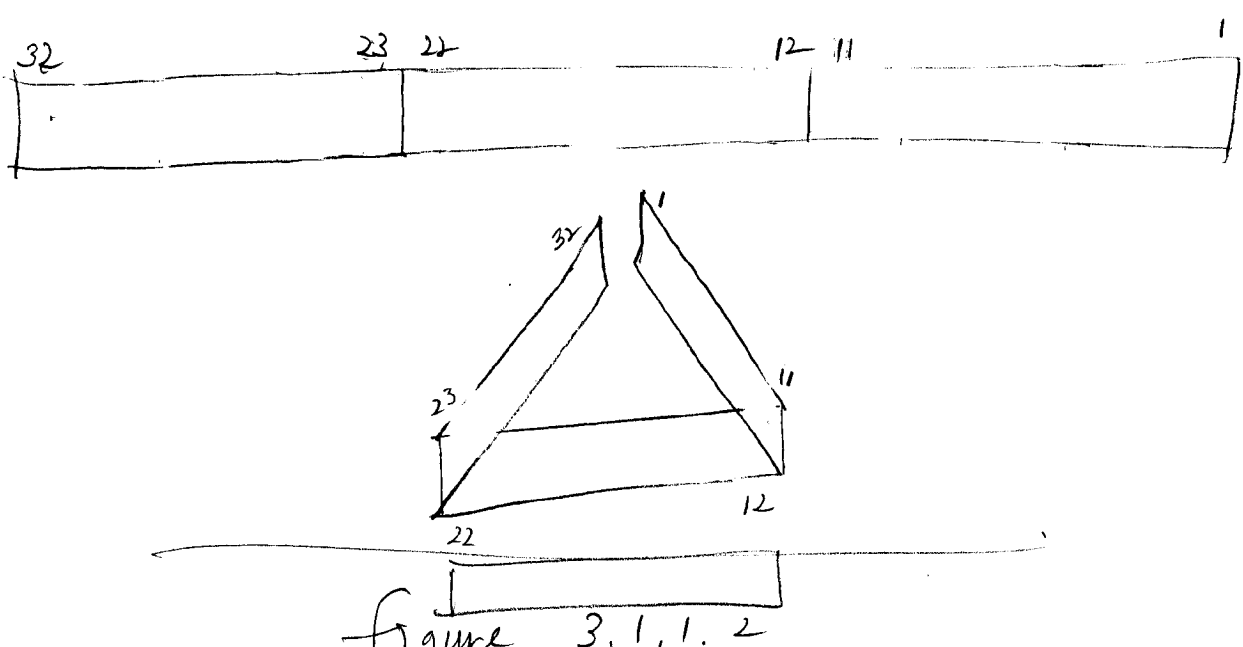
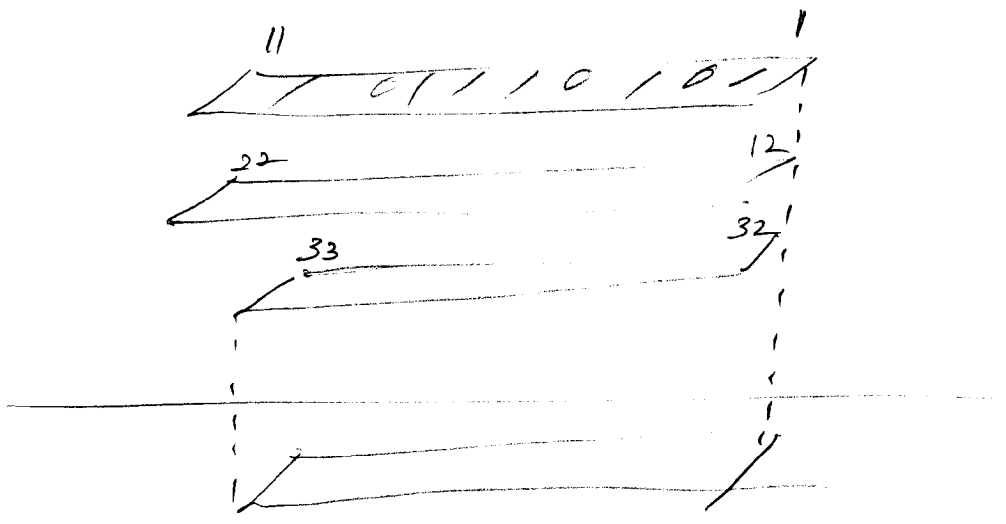


figure 3.1.1.2

Then K bits in the resulting partition are used to represent a hash address. This folding method is specifically called fold-boundary (or folding at the boundaries).

If 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 as shown in figure 2.1.1.3, this method is often referred to as fold-shifting (or shift folding).

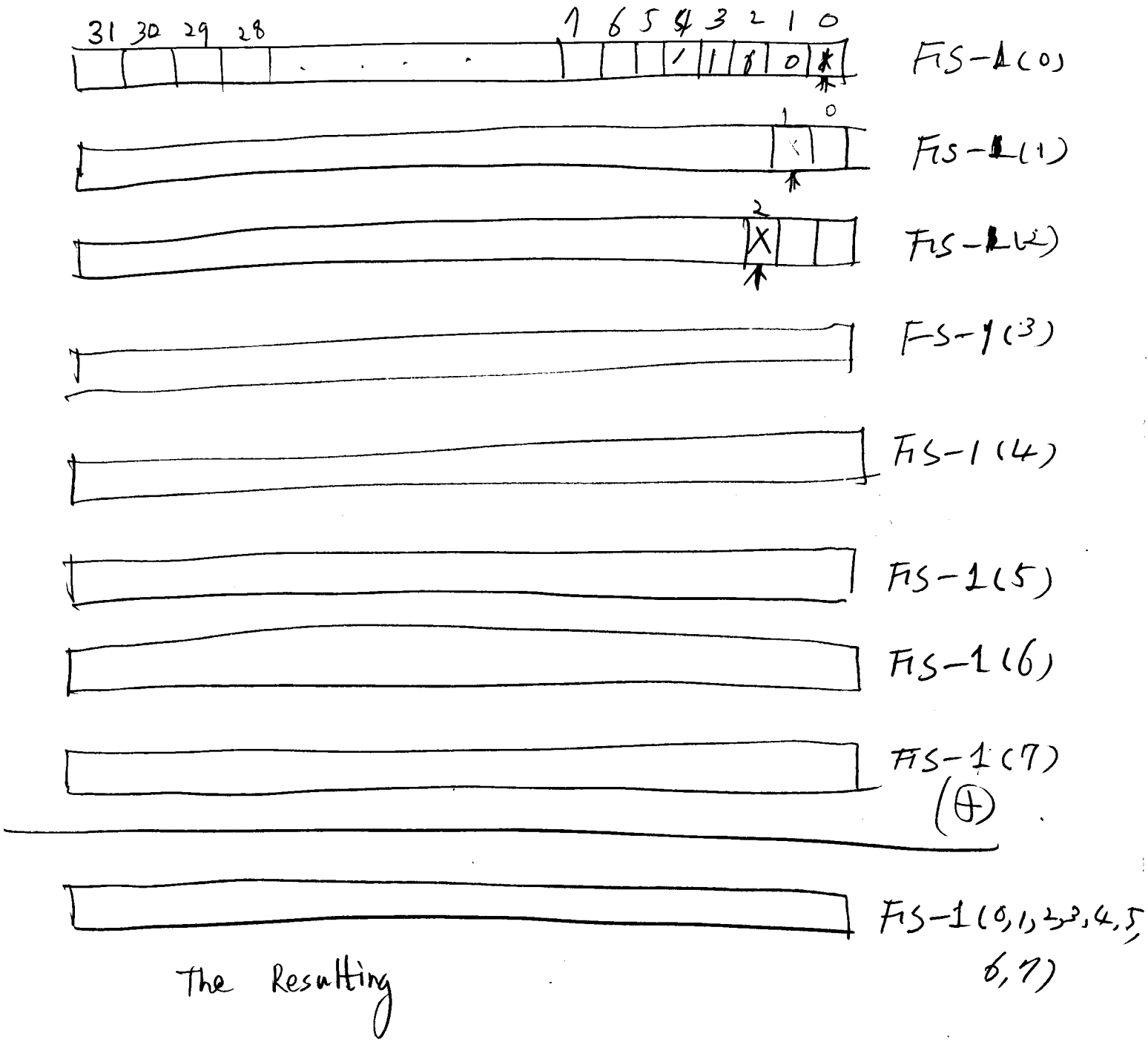
Fold-Shifting

This idea has been discussed by several researchers <MAUR1, KNUT1, KNOT1, LUM1>. When there is an (encoded) one word key, there may be many ways to fold using exclusive-OR operation. The questions are how many bits should be shifted and how many times of folding processes are necessary to provide good enough distribution of the keys. It is, however, clear that it is not a good idea to load an intermitten result into a register for a shift in next step because repetitive loading, exclusive-ORing, and shifting processes will not help producing a hash address within few machine cycle. Therefore, the original encoded key has to be shifted not by a shift register, but by wires which shiftedly connected to exclusive-OR gates.

In answering the above questions, it is necessary to consider how many shifted key words are needed in folding to randomize the bits in the resulting word. It is easy to

see that there are similar patterns in an (encoded) key since the bit patterns of ASCII still affects each bits in the encoded key. In other words, each byte in an encoded key might have a similar pattern. This pattern in each byte should be eliminated in folding process. So the scope of randomization is narrowed down to a byte. If the number of bits rotated is one, then 8 rotated key words might be sufficient to randomize every bits in a byte. This fold-shifting process represented with FS-1(0,1,2,3,4,5,6,7) is shown in Figure 3-8. If the number of bits rotated is two, then 4 rotated key words might be enough in randomizing every bits in a byte. For example, FS-2(0,2,4,6) as shown in Figure 3-9 is equivalent to any combination of (0,2,4,6) in FS-2 (e.g. FS-2 (2,4,6,0), FS-2(4,6,0,2), and etc.). And FS-2(0,2,4,6) is symmetric to FS-2(1,3,5,7) since their resulting bits are only ordered differently. Here, one can see the general idea that the

number of rotated key words required is the upper bound of the number of bits in a byte (8) / the number of bits rotated. For hardware implementation, it would be better if the number of rotated key words is 2^k ($K=1, 2,$ or 3) due to even number of input lines in a exclusive-OR gate. So for FIS-3, one might have FIS-3(0,3,6,1). And for FIS-4, FIS-4(0,4,1,5) can be used instead of FIS-4(0,4) or FIS-4(0,4,0,4). For FIS-5, FIS-5(0,5,2,7) can be used instead of FIS-5(0,5). While investigating FIS's, one might suggest that the number of rotated key words should be 4 (2**k, k=2) since 2 is too little and 8 is too many. Based on that idea, FIS-6(0,6,4,2) can be selected for FIS-6, but it is equivalent to FIS-2(0,2,4,6). So FIS-6(0,6,4,2) is not considered. Now FIS-7 (0,7,6,5) can be a representative for FIS-7.



The Resulting

Figure 3-8 FS_1(0,1,2,3,4,5,6,7) process.

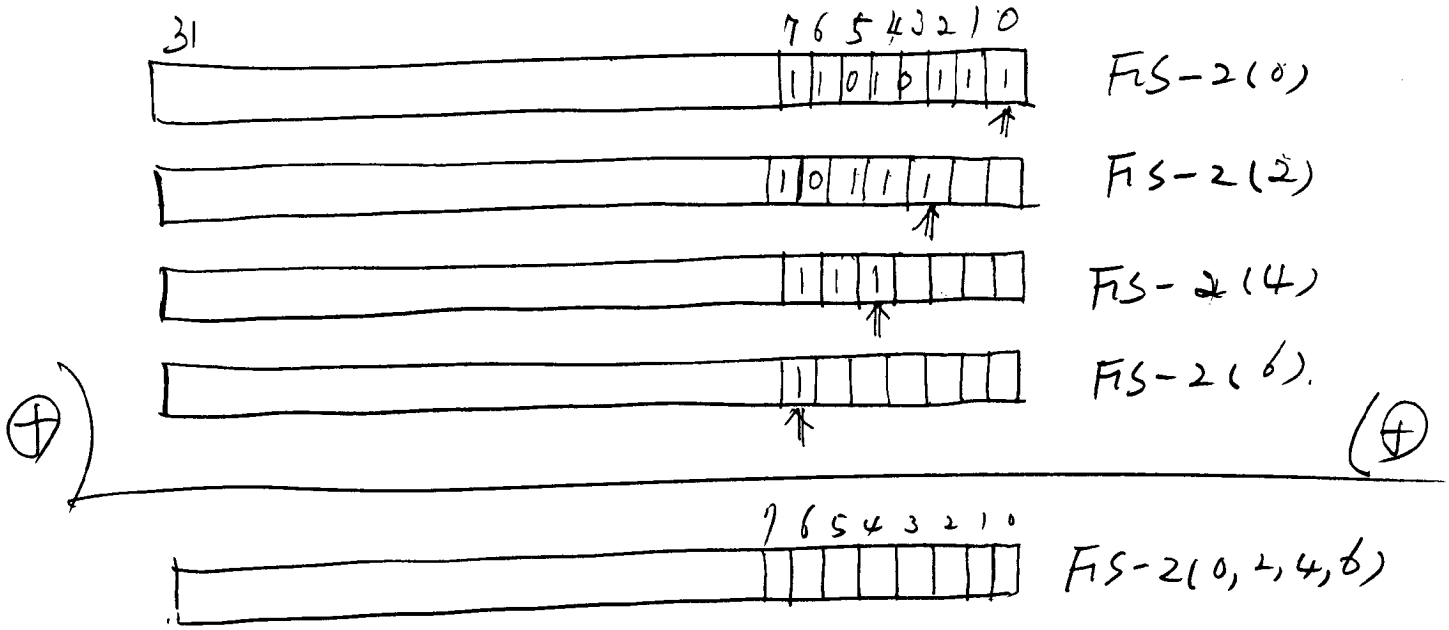


Figure 3-9 FIS-2 (0, 2, 4, 6) process.

Consequently the fold-shiftings to be experimented are FS-2(0,2,4,6), FS-3(0,3,6,1), FS-4(0,4,1,5), FS-5(0,5,2,7), and FS-7(0,1,6,5). Their distribution performances are discussed in the next section.

The Midsquare Method

In the midsquare hash method, 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, the range of hash values is from 0 to 2**k-1. The number of buckets in the hash table must be a power of 2 (e.g. 2**k) when this sort 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.

Multiplicative Hash Method

In this hash method, a real number C between 0 and 1 is chosen. The hash function is defined as truncate(m * fraction(C * key)), where fraction(x) (= x - truncate(x)) is the fractional part of the real number x. In other words, the key is multiplied by a real number (C) between 0 and 1, the fractional part of the product is taken to provide a random number between 0 and 1 dependent on every bit of the key, and multiply 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**5)* C is an integer relatively prime to 2**5. (C should not be too close to either 0 or 1). Also if r is the number of possible character codes, avoid values C such that fraction(r^k * C) is

too close to 0 or 1 for some small value of k and values c of the form $i/(r-1)$ or $i/(r^2-1)$. Values of c that yield good theoretical properties are .6180339887, which equals (sqrt(5) - 1)/2, or .3819660113, which equals 1 - (sqrt(5) - 1)/2.

If m is chosen as a power of 2 such as 2^p, the hash addresses can be calculated efficiently by multiplying the one-word integer key by the one-word integer $c * 2^b$ to produce a two-word product. The integer represented by the most significant p bits of the integer in the second word of this product is then equal to the hash value.

```
function Hash_Multiplicative (Key_Value : integer) :
                                                integer;
const
    C_Value = 0.6180339887

Var
    fraction, temp : real;
begin
    temp := C_Value * key_Value;
    fraction := temp - trunc(temp);
    Hash_Multiplicative := trunc( No_Buckets * fraction);
end;
```

The Radix Method

In this hash method, a number representing the key is considered as a number in some selected base (e.g., base 11) than in it's real base (e.g., base 8). Then the resulting number is converted to base 10 for a decimal address. For example, the key 7286 in base 10 is considered as 7286 in base 11, so 7286 in base 11 becomes 9653 in base 10. as shown in the equation below.

$$7 \times 11^3 + 2 \times 11^2 + 8 \times 11^1 + 6 = 9653 \text{ (in base 10)}$$

The number 9653 can be divided by the number of buckets in the hash table to use the remainder as a hash address just like in division method. ✓ Or ① next page the number 9653 can be multiplied by some fraction and the fractional part of the product is truncated to use a number within the range of the hash table as a hash address. The idea in this method is that converting a number in one base to another hashes up the bits, and the hashed

value is used to produce a hash address.

① This combination of two methods : radix transformation and division methods is originated from Lin's work.

The Random Method

This hash method uses a statistically approved pseudo-random number generating function which assumingly generates a sequence of random numbers using a provided starting number (or the seed). After the key is encoded, the encoded word W is inputted to the random number generating function as the seed. Then apply division or some other method to the generated random number to produce a hash address.
The distribution performance of this hash function might be dependent on the chosen pseudo-random number generating function.

Pearson's PC hash Method.

Recently Pearson introduce a new hash method for the personal computers which do not provide fast 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 3.3.1.2, An auxiliary table (T) is used to contain 256 randomish integers range from 0 to 255. The Pearson's hash function receives a string of characters in ASCII code. Each character (C[i]) is represented by one byte which is used as an index in the range 0-255. As shown in the Pearson's hash algorithm, each character of a key is exclusive-ORed with an indexed memory read (h[i-1]). The resulting byte is used to index the table T, and the indexed value in T is stored to h[i] for the next iteration step. After the looping process is finished,

```
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 2.3.1.#

the last indexed value (h[n]) from the table T is the hash address for the buckets ranging 0 through 255..

Pearson <PEAR1> claimed that it is not necessary to know the length of the string at the beginning of the computation, a property useful when the end of the text string is indicated by a special character rather than by a separately stored length variable.
He added that one can generate his own random permutations for the table (T).

Pseudorandom Permutation of the integers 0 through 255

1
14
110
25

figure

3.4 An Analysis of Distribution, Speed, and Cost

As shown in the table 1, each hash function's distributions on the three different data sets, speed when implemented either in software or in hardware, and cost of the hardware implementation of the hash function. For the measurement of distributions, mean square deviations are provided. The number of clock cyles is used in the measurement of the speeds of the hash coders. The cost of building a hardware hash coder is represented with the number of gates needed. Performances of each hash function are discussed based on the experiment results.

section 3.4

Hash Methods (D=Divisor)	<Distribution Performance> (Mean Square Deviation) When Applied to Three Different Data Sets: Randomly Chosen Names	Generally Chosen Names	Randomly Chosen Numeric Strings	<Speed> (Clock Cycles) When Implemented in Software	Hardware	<Cost> (Gates)
Mapping	~~3.93~~ 4.16 (Avr.)	3.96 (Avr.)	3.79 (Avr.)	~~126~~ 96	3	120 &
Mapping with Additions	4.40	3.91	3.58	96	64	182
Division with Additions	3.97	3.91	86.76	86	76 47 **	294 3478
Maurer's Shift and Fold	3.95 (Avr.)	4.06 (Avr.)	3.81 (Avr.)	420	70	~~640~~ 3.4
Algebraic Coding(D=1021)	4.41	4.62	3.77	452	48	390
Berkovich's Hu_Tucker Code	4.09	3.97	3.70	826 @	66 @	399
Digit Analysis (2 & 4 bytes)	4.32 3.80	4.07 4.70	3.84 19.74	≥(40)+	2 +	112 96
Division (D=257)	5.67	11.95	122.99	70	46 16 **	390 3360
Folding	4.09	3.89	53.02	56	2	117
Midsquare	4.25	4.84	88.91	72	30 8 *	572 2796
Multiplicative	4.42	3.29	12.49	407	64 17 *	422 2892
Radix	3.97	4.05	12.36	650	390 285 * 120 **	550 3234 6498
Random	4.25	3.63	9.79	162	80 57 * 26 **	470 3138 6402

+ pre-analysis is required beforehand

& Sixteen 64x16 bits RAMs are also required.
* Faster but expensive since Wallace Tree is used for multiplication.
** Faster but expensive since division array is used for division.
@ Slightly changeable due to variable length encoded key string.

Pearson

Table 1 -1-

82 82 280

105

3.4.1 Performance of Mapping Hash

Distribution performances of mapping hash method have been measured in two different cases: each RAM contains identical set of 64 words of prime numbers, and each RAM contains different set. The table 3.2 shows the distribution performances in terms of mean square deviation (MSD) for both cases when the mapping hash method is applied to the three different data sets such as randomly and generally chosen names, and randomly chosen numeric strings.

The MSDs of mapping hash method when RAM contents are identical.

Data Set \ Bits Selected	2–9	3–10	4–11	5–12	6–13	Avr.
Random Names	4.34	4.15	4.22	4.14	3.95	4.16
General Names	3.29	3.45	4.36	4.28	4.40	3.96
Numeric Strings	3.179	3.79	3.87	3.87	3.64	3.79

The MSDs of mapping hash method when RAM contents are different

Data Set \ Bits Selected	2–9	3–10	4–11	5–12	6–13	Avr.
Random Names	3.74	3.83	4.13	4.20	3.77	3.93
General Names	4.41	4.54	3.91	3.83	3.60	4.06
Numeric Strings	3.95	4.05	4.40	3.74	4.22	4.07

Table 3.2

The prime numbers in the RAMs are arbitrally chosen without help of a random generator, so the distribution performances of the mapping hash in the table 3.1 might be improved if an appropriate bucket tuning method is carried out to provide a proper set of prime number for the RAMs.

As shown in the table 3.2, mean square deviations are hovering around 4 as those of other relatively good hash methods'. The results does not give any clue of data dependency since the mapping hash function distributes numeric string keys as well as other keys. It is also clear to say that there is no distinguishable difference in having RAM contents different or identical. Different groups of 8 bits (i.e., 2-9; 3-10, 4-11, 5-11, 6-13 bits) are extracted to compose hash addresses, and there is no noticeable difference

between the distributions of groups.

Our major objective in developing a new hash method is extremely fast hash address calculation using any necessary hardware aids. By virtue of byte by byte parallel processing with separate RAM and 4 levels of exclusive-OR gates for each character, the mapping hash method can produce a hash address within 3 clock cycles; Two clock cycles are required for the memory read to retrieve a random number from the coresponding RAM$_x$ as specified in <MOTO1> One clock cycle is taken for the calculation process for hash address bits through the 4 levels of exclusive-OR gates. As mentioned previously, the maximum gate delay is 9 nanoseconds and the clock frequency is set to 20 MHz (50 nanoseconds per a clock pulse width), so the address bit signal can pass through the 4 gate levels ($4 * 9 = 36 \leq 50$ nsecs)

within a clock cycle. If this hash coder is implemented in software, the speed (126 clock cycles) is more than 40 times slower than the hardware hash coder (3 clock cycles).

The number of gates needed to implement the hardware mapping hash coder is 120 since 8 exclusive-OR modules (8 x 15 = 120) are required to produce 8 bits for a hash address. In addition to the gates, sixteen 64 x 16 bits (64 prime numbers) of RAMs are also required to convert 16 characters in a key to 16 corresponding prime numbers in respective RAMs.

The performance of the additive mapping hash method is also examined. The mean square deviations of the additive hash method are 4.40, 3.91, and 3.58 when it is applied to the randomly chosen names, generally chosen names, and randomly chosen numeric strings data sets respectively. The additive mapping hash method shows competitive distribution performances. This result support the claim that addition and exclusive-OR ing gives same effect in randomizing the bit values. However, the hardware implementation of the additive mapping hash method does not substantially speed up the hash address calculation time (It still takes 64 clock cycles) as hardware implementation improves in the exclusive-OR mapping hash method. The speed of software implemented additive mapping hash method is the same with that of software implemented

exclusive-OR mapping hash method (96 clock cycles) based on MC68030 instructions' execution time. the table of.

The architecture of the additive mapping hash coder is the same with that of carry lookahead adder <CAVA1>. which is faster than the ripple adder. The number of gates needed to develop 16 bits carry lookahead adder is 182. And sixteen 64x16 bits RAMs as used in the hardware mapping hash coder are not necessary since in serial processing. One set of 64 prime numbers can be stored in main memory instead.

3.4.2 Performance of Maurer's Shift and Fold Method

The distribution performance of Maurer's shift and fold hash method is measured in 10 different groups of 8 bits among 128 resulting bits. The first group of 8 bits for a hash address is made up of 10th through 17th resulting bits from the hash method. The second group is composed of 20th through 27th resulting bits from the hash method. The other groups are made up like 30th through 37th (30-37), 40-47, 50-57, 60-67, 70-77, 80-87, 90-97, and 100-107 resulting bits for hash address. The mean square deviations of these groups are computed by having the hash method applied to the three data sets such as randomly chosen names (RCN), generally chosen names (GCN), and randomly chosen numeric strings (RNS), and shown in table 3.2. The measured mean square deviations

Bits	10-17	20-27	30-37	40-47	50-57	60-67	70-77	80-87	90-97	100-107
RCN	4.29	3.27	3.79	4.13	3.66	4.19	3.63	4.13	4.25	4.13
GCN	4.13	3.84	4.66	4.11	3.71	4.19	4.52	3.80	3.91	4.32
RNS	3.88	4.04	4.45	4.07	3.81	3.66	4.12	4.01	3.94	3.74

Data set	Average of 10 MSPs
RCN	3.95
GCN	4.12
RNS	3.97

Table 3.2 Distribution Performance of Mourer's Shift and Fold Hash Method.

are mostly between 3 and 5 regardless of the selected resulting bits. This result assures that any of the resulting bits generated by this hash method can be included in producing a hash address.

The speeds of this hash method is 420 and 70 clock cycles when it is implemented in software and hardware. Since the key registers for 16 bytes cannot be assumed that they are connected together in a conventional processor, the software implementation of this hash method takes considerable time in rotating bits in a word register and copying some bits to the next word register. Therefore, this hash method is also a hardware oriented hash method like the mapping hash method. When this method is implemented in hardware, one lengthy shift register for a 4 words key is provided for a fast shift operation. This hash method in hardware uses

fast operations such as shift and exclusive-OR, but the speed (10 clock cycles) is not prominent. The hindrance in speed is caused by the log n times (n is the number of bits in a key, so log 128 = 7) of loading operation. This loop in the hash algorithm may help better distribution of keys, but reduce the performance in speed.

The cost of this hardware coder is counted as 384 gates which include flip-flops (2 gates each) in the shift register and exclusive-OR gates.

3.4.3 Performance of Berkovich's Hu_Tucker Code Hash Method

As shown in table 1, the mean square deviations (4.09, 3.97, and 3.77) show that the distribution performance of Berkovich's Hu_Tucker code hash method is first-class. Seemingly, this hash method is not data dependent since there is no distinguishable difference in the three mean square deviations. This data independency is perhaps resulted from the variable length of the Hu-Tucker code for each character. While a Hu_Tucker coded key is folded several times at the length of hash address bits, another randomization is added onto the Hu_Tucker code conversion which already randomized the value of the key in some degree.

The speed of this hash code is dependent

on the length of the coded bit string. For example, the Hu-Tucker coded value of the key 'XXX' is '11111100 11111100', whereas that of the key 'EEE' is '010 010 010'. Hence, the key 'XXX' requires more folding steps which results in more time in hash address calculation than the key 'EEE' does. Therefore, the speeds (826 and 128 clocks) which are shown in the table 1 can be increased or decreased depending on every input key. Those clock cycle figures are obtained using some medium size Hu-Tucked coded bit string. The major drawback in the speed performance of this hash method is adding the number of bits in a Hu-Tucker code each time to get the cumulative number of bits in the encoded bit string.

The cost of the hardware hash coder is analyzed such that 399 gates are needed on the assumption that

8 bits are the maximum number of bits in Hu-Tucker codewords. The register which contains the encoded bit string may have 128 flip-flops (8 bits in each of 16 characters). And a 8 bit counter register and several levels of exclusive-OR gates are helpful in speeding up the address calculation.

3.4.4 Performance of the Algebraic Coding Hash Method

The Galois Field GF(2) has been chosen to analyze the performance of the Algebraic Coding Hash Method. The distribution performance of this method (MSDs : 4.41, 4.62, 3.77) as shown in table 1 is judged as top-rate, and it doesn't reveal data dependency. Even though this hash method is experimented under a certain codition, by the result, it is convincing that the hash method may provide relatively good distribution under other circumstances. Although this is a hardware oriented hash method, the speed performance (48 clock cycles) in hardware implementation is not beyond our expectation. If this method is implemented in hardware, it is relatively slow (452 clock cycles). And the number of gates needed to implement this method in hardware is approximately 390.

3.4.5 Performance of the Digit Analysis Hash Method

The distribution performance of the digit analysis hash method is measured using two types of encoded key (2 bytes and 4 bytes). Before keys are being hashed, the key are scanned to provide information about the distribution of values of a key in each digit. The resulting statistics are shown in table 3, 4, and 5 for the respective data sets (RCN, GCN, and RNS). The most skewed distributions (digits with a large deviation) are deleted until the number of remaining digits is equal to the hash address length (8 bits). One might say that if the hash address bits consist of the selected bits from the digit analysis, the distribution performance must be good and better than other selection of digits. After an investigation,

it is found that the selected group of bits for hash address in this hash method is not always the best group of bits. As shown in table 1, In the case of 4 bytes encoded key, the MSD of RNS is 19.74 which is relatively poor distribution performance. However, the MSD's in case of 2 bytes encoded key are 4.32, 4.07, and 3.84, so it is hesitant to say that this hash method is data dependent.

Since the keys in input data file have to be scanned twice (one for digit analysis and one for hashing), the speed of this hash method (40 clock cycles for software implementation and 2 clock cycles for hardware implementation) cannot be compared with others. When a static input file is read and digits are selected by an analysis, the keys in that static file are hashed very fast. Thus, this hash method is good for a static input file.

The cost to build this hash coder in hardware is the cheapest (either 112 or 96 gates majorly for encoding) since it does not include any hardware function for mathematical operation.

STATISTICS FOR DIGIT ANALYSIS HASH METHOD
WHICH IS APPLIED TO THE 1024 RANDOMLY CHOSEN NAMES DATA

Number of 1s in Each Digit in Two Bytes Key

Bit No. 1	489	1s	---	47.75%	DEVIATION :	529
Bit No. 2	533	1s	---	52.05%	DEVIATION :	441
Bit No. 3	502	1s	---	49.02%	DEVIATION :	100
Bit No. 4	484	1s	---	47.27%	DEVIATION :	784
Bit No. 5	548	1s	---	53.52%	DEVIATION :	1296
Bit No. 6	585	1s	---	57.13%	DEVIATION :	5329
Bit No. 7	519	1s	---	50.68%	DEVIATION :	49
Bit No. 8	0	1s	---	0.00%	DEVIATION :	262144
Bit No. 9	501	1s	---	48.93%	DEVIATION :	121
Bit No. 10	534	1s	---	52.15%	DEVIATION :	484
Bit No. 11	498	1s	---	48.63%	DEVIATION :	196
Bit No. 12	524	1s	---	51.17%	DEVIATION :	144
Bit No. 13	500	1s	---	48.83%	DEVIATION :	144
Bit No. 14	420	1s	---	41.02%	DEVIATION :	8464
Bit No. 15	486	1s	---	47.46%	DEVIATION :	676
Bit No. 16	0	1s	---	0.00%	DEVIATION :	262144

Digits 2, 3, 7, 9, 10, 11, 12, 13 are selected for Digit Analysis hash method.

Number of 1s in Each Digit in One Word Key

Bit No. 1	536	1s	---	52.34%	DEVIATION :	576
Bit No. 2	508	1s	---	49.61%	DEVIATION :	16
Bit No. 3	498	1s	---	48.63%	DEVIATION :	196
Bit No. 4	500	1s	---	48.83%	DEVIATION :	144
Bit No. 5	473	1s	---	46.19%	DEVIATION :	1521
Bit No. 6	689	1s	---	67.29%	DEVIATION :	31329
Bit No. 7	435	1s	---	42.48%	DEVIATION :	5929
Bit No. 8	0	1s	---	0.00%	DEVIATION :	262144
Bit No. 9	514	1s	---	50.20%	DEVIATION :	4
Bit No. 10	467	1s	---	45.61%	DEVIATION :	2025
Bit No. 11	516	1s	---	50.39%	DEVIATION :	16
Bit No. 12	481	1s	---	46.97%	DEVIATION :	961
Bit No. 13	430	1s	---	41.99%	DEVIATION :	6724
Bit No. 14	329	1s	---	32.13%	DEVIATION :	33489
Bit No. 15	483	1s	---	47.17%	DEVIATION :	841
Bit No. 16	0	1s	---	0.00%	DEVIATION :	262144
Bit No. 17	519	1s	---	50.68%	DEVIATION :	49
Bit No. 18	505	1s	---	49.32%	DEVIATION :	49
Bit No. 19	512	1s	---	50.00%	DEVIATION :	0
Bit No. 20	520	1s	---	50.78%	DEVIATION :	64
Bit No. 21	483	1s	---	47.17%	DEVIATION :	841
Bit No. 22	366	1s	---	35.74%	DEVIATION :	21316
Bit No. 23	502	1s	---	49.02%	DEVIATION :	100
Bit No. 24	0	1s	---	0.00%	DEVIATION :	262144
Bit No. 25	501	1s	---	48.93%	DEVIATION :	121
Bit No. 26	509	1s	---	49.71%	DEVIATION :	9
Bit No. 27	522	1s	---	50.98%	DEVIATION :	100
Bit No. 28	493	1s	---	48.14%	DEVIATION :	361
Bit No. 29	418	1s	---	40.82%	DEVIATION :	8836
Bit No. 30	353	1s	---	34.47%	DEVIATION :	25281
Bit No. 31	501	1s	---	48.93%	DEVIATION :	121
Bit No. 32	0	1s	---	0.00%	DEVIATION :	262144

Digits 2, 9, 11, 17, 18, 19, 26, 27 are selected for Digit Analysis hash method.

Table 3

STATISTICS FOR DIGIT ANALYSIS HASH METHOD
WHICH IS APPLIED TO THE 1024 NUMERIC STRINGS DATA

Number of 1s in Each Digit in Two Bytes Key

Bit No. 1 :	502	1s ---	49.02%	DEVIATION :	100
Bit No. 2 :	521	1s ---	50.88%	DEVIATION :	81
Bit No. 3 :	478	1s ---	46.68%	DEVIATION :	1156
Bit No. 4 :	517	1s ---	50.49%	DEVIATION :	25
Bit No. 5 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 6 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 7 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 8 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 9 :	516	1s ---	50.39%	DEVIATION :	16
Bit No. 10 :	529	1s ---	51.66%	DEVIATION :	289
Bit No. 11 :	518	1s ---	50.59%	DEVIATION :	36
Bit No. 12 :	492	1s ---	48.05%	DEVIATION :	400
Bit No. 13 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 14 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 15 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 16 :	0	1s ---	0.00%	DEVIATION :	262144

Digits 1, 2, 3, 4, 9, 10, 11, 12 are selected for digit analysis hash method.

Number of 1s in Each Digit in One Word Key

Table 4

Bit No. 1 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 2 :	512	1s ---	50.00%	DEVIATION :	0
Bit No. 3 :	509	1s ---	49.71%	DEVIATION :	9
Bit No. 4 :	474	1s ---	46.29%	DEVIATION :	1444
Bit No. 5 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 6 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 7 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 8 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 9 :	512	1s ---	50.00%	DEVIATION :	0
Bit No. 10 :	522	1s ---	50.98%	DEVIATION :	100
Bit No. 11 :	514	1s ---	50.20%	DEVIATION :	4
Bit No. 12 :	436	1s ---	42.58%	DEVIATION :	5776
Bit No. 13 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 14 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 15 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 16 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 17 :	502	1s ---	49.02%	DEVIATION :	100
Bit No. 18 :	509	1s ---	49.71%	DEVIATION :	9
Bit No. 19 :	495	1s ---	48.34%	DEVIATION :	289
Bit No. 20 :	443	1s ---	43.26%	DEVIATION :	4761
Bit No. 21 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 22 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 23 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 24 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 25 :	500	1s ---	48.83%	DEVIATION :	144
Bit No. 26 :	505	1s ---	49.32%	DEVIATION :	49
Bit No. 27 :	526	1s ---	51.37%	DEVIATION :	196
Bit No. 28 :	420	1s ---	41.02%	DEVIATION :	8464
Bit No. 29 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 30 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 31 :	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 32 :	0	1s ---	0.00%	DEVIATION :	262144

Digits 2, 3, 9, 10, 11, 17, 18, 26 are selected for digit analysis hash method.

STATISTICS FOR DIGIT ANALYSIS HASH METHOD
WHICH IS APPLIED TO THE 1024 GENERALLY CHOSEN NAMES DATA

Number of 1s in Each Digit in Two Bytes Key

Bit No. 1	483	1s ---	47.17%	DEVIATION :	841
Bit No. 2	502	1s ---	49.02%	DEVIATION :	100
Bit No. 3	487	1s ---	47.56%	DEVIATION :	625
Bit No. 4	515	1s ---	50.29%	DEVIATION :	9
Bit No. 5	513	1s ---	50.10%	DEVIATION :	1
Bit No. 6	505	1s ---	49.32%	DEVIATION :	49
Bit No. 7	500	1s ---	48.83%	DEVIATION :	144
Bit No. 8	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 9	506	1s ---	49.41%	DEVIATION :	36
Bit No. 10	523	1s ---	51.07%	DEVIATION :	121
Bit No. 11	526	1s ---	51.37%	DEVIATION :	196
Bit No. 12	542	1s ---	52.93%	DEVIATION :	900
Bit No. 13	504	1s ---	49.22%	DEVIATION :	64
Bit No. 14	494	1s ---	48.24%	DEVIATION :	324
Bit No. 15	472	1s ---	46.09%	DEVIATION :	1600
Bit No. 16	0	1s ---	0.00%	DEVIATION :	262144

Digits 2, 4, 5, 6, 7, 9, 10, 13 are selected for Digit Analysis hash method.

Number of 1s in Each Digit in One Word Key

Bit No. 1	492	1s ---	48.05%	DEVIATION :	400
Bit No. 2	520	1s ---	50.78%	DEVIATION :	64
Bit No. 3	518	1s ---	50.59%	DEVIATION :	36
Bit No. 4	517	1s ---	50.49%	DEVIATION :	25
Bit No. 5	475	1s ---	46.39%	DEVIATION :	1369
Bit No. 6	744	1s ---	72.66%	DEVIATION :	53824
Bit No. 7	429	1s ---	41.89%	DEVIATION :	6889
Bit No. 8	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 9	522	1s ---	50.98%	DEVIATION :	100
Bit No. 10	517	1s ---	50.49%	DEVIATION :	25
Bit No. 11	510	1s ---	49.80%	DEVIATION :	4
Bit No. 12	498	1s ---	48.63%	DEVIATION :	196
Bit No. 13	462	1s ---	45.12%	DEVIATION :	2500
Bit No. 14	275	1s ---	26.86%	DEVIATION :	56169
Bit No. 15	443	1s ---	43.26%	DEVIATION :	4761
Bit No. 16	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 17	509	1s ---	49.71%	DEVIATION :	9
Bit No. 18	512	1s ---	50.00%	DEVIATION :	0
Bit No. 19	535	1s ---	52.25%	DEVIATION :	529
Bit No. 20	516	1s ---	50.39%	DEVIATION :	16
Bit No. 21	424	1s ---	41.41%	DEVIATION :	7744
Bit No. 22	399	1s ---	38.96%	DEVIATION :	12769
Bit No. 23	515	1s ---	50.29%	DEVIATION :	9
Bit No. 24	0	1s ---	0.00%	DEVIATION :	262144
Bit No. 25	502	1s ---	49.02%	DEVIATION :	100
Bit No. 26	518	1s ---	50.59%	DEVIATION :	36
Bit No. 27	522	1s ---	50.98%	DEVIATION :	100
Bit No. 28	496	1s ---	48.44%	DEVIATION :	256
Bit No. 29	462	1s ---	45.12%	DEVIATION :	2500
Bit No. 30	335	1s ---	32.71%	DEVIATION :	31329
Bit No. 31	545	1s ---	53.22%	DEVIATION :	1089
Bit No. 32	0	1s ---	0.00%	DEVIATION :	262144

Digits 4, 10, 11, 17, 18, 19, 23, 26 are selected for Digit Analysis hash method.

Table 5.

3.4.6 Performance of the Division Hash Method

The distribution performance of the division hash method varies depending on the chosen divisor which is close to the number of buckets. as shown in table 6. If an inappropriate divisor is chosen, the data dependency problem can severely occur. As Buchholz suggested, using the largest prime number smaller than the number of buckets as the divisor is the safest bet. The divisor 257 is a nonprime number with prime factors less than 20 as recommended by LUM et. al, but it shows very poor distributions (MSDs : 5.67, 11.95, and 122.99).

The speeds of the division method are relatively fast in both software implementation (70 clock cycles) and hardware implementation (40 or 16 clock cycles). The two major ways of

implementing the division hash method in hardware are sequential shift-subtract/add non-restoring (or restoring) division and division array <CAPP1>. By adapting the division array, the speed of division operation can be reduced from 46 clock cycles to 16 clock cycles, but the cost of the hardware is increased from 390 gates (sequential shift-subtract/add nonrestoring division) to 3360 gates.

The performance of the additive division method (add the ordinal number of each character in a key and divide the sum by the number of buckets) has been also analyzed. (MSD's are 3.97, 3.91, and 86.76). The poor distribution (MSD 86.76) has been resulted when the additive division is applied to the data set RNS when the cumulative numbers (or SUMs) are not large enough comparing from the numeric character strings

With the number of buckets. Therefore, the additive division method can be used only when the average ratio of sums and the number of buckets (sums/number of buckets) is sufficient.

Several other researchers <BUCH1, LUM1; RAMA1> conducted experiments on typical key sets searching for the ideal hashing. Their overall conclusion is that the simple method of division seems to be the best general key to address transformation technique when computational time is not heavily considered. 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 depending on the application environment. Especially, in the application where

fast hash address calculation is not required, the additive mapping method seems to be superior to the division method since using the additive mapping method, one does not need to worry about selecting a right divisor and just divides the sum (or combination) by the number of buckets to get a remainder for a hash address. On the other hand, where the speed in address calculation is imperative and the number of buckets can be 2**n so that a hardware hash coder is needed, the mapping hash coder which is much faster and cheaper than the division hash coder is recommended.

3.4.7 Performance of Folding Hash Method

There might be several different types of folding method. In this experimental environment, fold-boundary method as described by Maurer <MAUR1> is simulated to show the distribution performance. The MSDs are 4.09, 3.89, and 53.02 when the bits (11th through 18th) which are exclusive-ORed twice are extracted to compose a hash address. As shown in table 6, the distribution performance on the randomly chosen numeric strings (RNS) data set is trifling. Bits like 12th through 19th and 13th through 20th are used to represent hash address, but data dependency is still reflected in the distribution. Some deliberately designed folding method (shift or rotate bits many different ways and fold them together) might prevent key distributions from this kind of data dependency problem.

3.4.8. Performance of the Midsquare Hash Method

The mean square deviations (4.25, 4.84, and 88.91) of the midsqure method tells that this method has dependency problem; therefore, in case that keys are similar in some form, this method has a high potential to give more key clusterings.

When this hash method is implemented in software, the hashing operation requires 72 clock cycles. The speed of hardware midsquare hash coder which uses sequential add shift multiplier <CAVA 1> (30 clock cycles) can be improved up to 8 clock cycles if Wallace's fast multiplier is used <WALL1>. However, the cost of the hardware hash coder increases from 572 gates to 2796 gates for the speed gain.

3.4.9 Performance of the Multiplicative Method.

By looking at the MSDs (4.42, 3.29, 12.49), the distribution performance of the multiplicative method is open to doubt whether it has inappreciable data dependency problem since the MSD (12.49) on the numeric strings (RNS) data set is not the one of the first-class hash method.

The software implement multiplicative hash coder is relatively slow (407 clock cycles), but Wallace's fast hardware multiplier built in the hash coder can improve the speed even upto 17 clock cycles with the cost of 2892 gates. The sequential add shift multiplier can be an alternative way of implementing the multiplicative hash coder. In this case, an address calculation requires 64 clock cycles and the cost of the hash coder is analyzed as 422 gates.

3.4.10 Performance of the Radix Hash Method

The distribution performance of the radix hash method is similar to that of the multiplicative hash method. The distribution performances (MSPs 3.97 and 4.05) of the names data set (RCN and GCN) are observably better than the distribution performance (MSD 12.36) of the numeric strings data set (RNS). So this hash method also has a potential to perform poorly when keys in a data set are similar.

The radix hash method is one of the most time consuming method since it takes a long time converting each digit of a key into another base number. The software implemented radix hash coder needs about 650 clock cycles to transform a key to an address. Because of the complexity in this hash algorithm, hardware implementation does not improve the speed (390, 285, and 120 clock cycles) as if increases the cost (550, 3234, and 6498 gates) as shown in table 1.

3.4.11 Performance of the Random Hash Method

The distribution performance of this hash method is dependent on the chosen random number generating function. In this experiment, the pseudo random number generating function suggested by Carta <CART1> is used in generating a hash address. The distribution performance (MSDs 4.25, 3.63, and 9.79) of this hash method is not the first-class. The speed performance when it is implemented in software is measured as 162 clock cycles which includes the computation times of a multiplication a division and an addition. The hardware hash coder for the random method can be implemented in many different ways using fast multiplier and hardware oriented divider as mentioned before.

An Economic hardware hashcoder which is not equipped with any of division array or Wallace's multiplier takes about 80 clock cycles in hash address calculation and requires 470 gates to have sequential multiplier and divider, carry lookahead adder, and exclusive-OR gates for encoding.

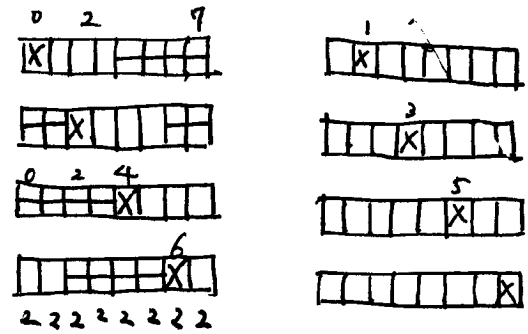
If the hash coden uses Wallace's fast multiplier, it can speed up the address computation to 57 clock cycles, but the cost will be raised upto 3138 gates. If the division array is added for fast division, the number of clock cycles in the address calculation is dropped to 26, but the expense is increased upto 6402 gates.

	RCN				GCN				RNS			
	1-8	9-16	17-24	25-32	1-8	9-16	17-24	25-32	1-8	9-16	17-24	25-32
(0,2,4,6) ⇒ (0,10,20,30)	4.48	3.88	4.20	3.84	4.77	3.63	3.96	4.27	3.44	4.58	4.41	3.56
			Avr.	4.1			Avr.	4.16				Avr. 3.99
(0,3,6,1) ⇒ (0,11,22,25)	3.73	4.23	4.03	3.34	4.34	3.71	4.46	4.14	3.51	4.19	4.88	3.94
			Avr.	3.83			Avr.	4.16				Avr. 4.13
(0,4,1,5) ⇒ (0,12,17,29)	4.23	4.12	4.41	4.02	4.43	4.15	4.98	3.14	4.02	4.98	20.70	3.69
			Avr.	4.20			Avr.	4.32			128, 0 bucket	Avr. 8.35
(0,5,2,7) ⇒ (0,13,18,31)	3.80	3.84	4.46	3.83	3.55	4.33	5.09	3.63	20.05	21.16	20.13	4.16
			Avr.	3.98			Avr.	4.15	129, 0 bucket			Avr. 16.38
(0,7,6,5) ⇒ (0,15,22,29)	4.07	4.11	4.05	4.42	4.17	4.13	5.28	4.05	53.02	20.64	21.60	21.00
			Avr.	4.16			Avr.	4.41			Avr.	29.07
(0,4,3,7) ⇒ (0,12,19,31)												

FS-2 (0, 2, 4, 6)

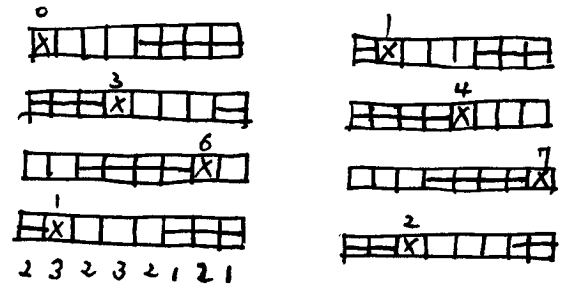
FS-2 (1, 3, 5, 7) X

FS-2 (



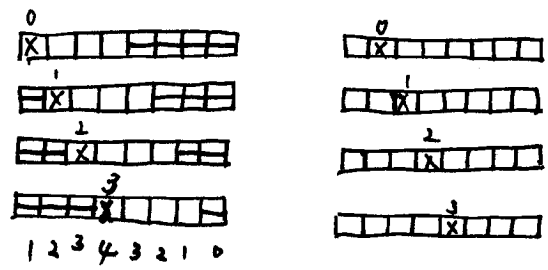
FS-3 (0, 3, 6, 1)

FS-3 (1, 4, 7, 2) (X)



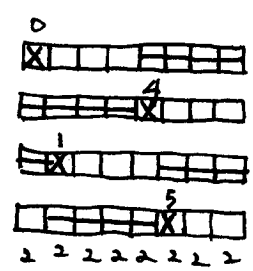
FS-1 (0, 1, 2, 3)

FS-2 (1, 2, 3, 4) (X)

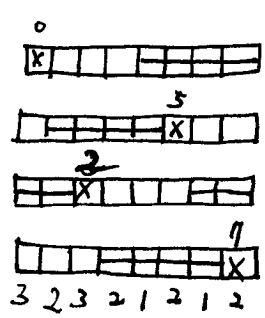


FS-4 (0, ~~4~~, 1, ~~5~~) No good

=FS-4 (1, 5, 0, 4) ☆.



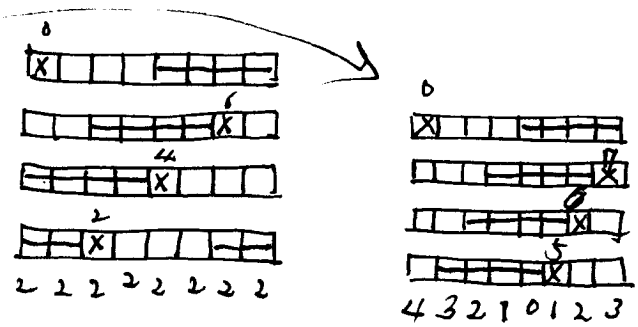
FS-5 (0, 5, ~~2~~, 7)

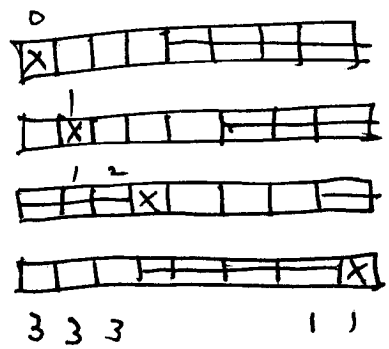


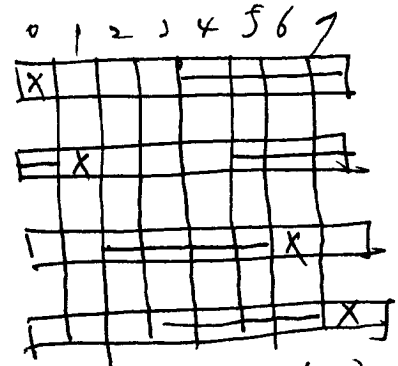
FS-6 (0, 6, 4, 2)

=FS-2 (0, 2, 4, 6)

FS-7 (0, 7, 6, 5)

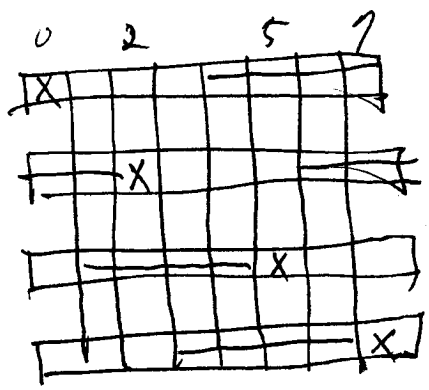


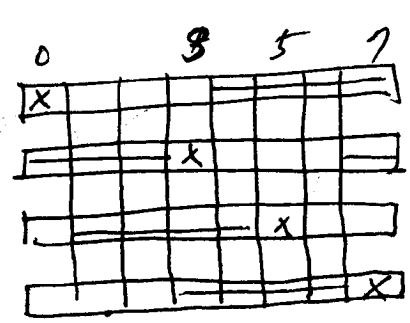

0
1
1 2
3 3 3
1 1


0 1 2 3 4 5 6 7
3 4 3 2 1 0 1 2

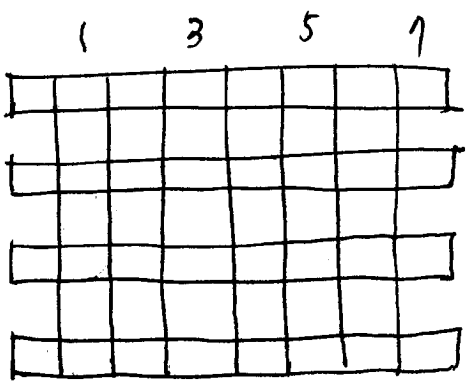
(X)

0, 1, 6, 7


0 2 5 7
3 3 2 2 1 2 1 2


0 3 5 7
3 2 2 2 1 2 2 2

(0, 2, 5, 7)


1 3 5 7

~~(2,2,4,6) ⇒ (0,10,20,30)~~	REN 1-8 9-16 17-24	GCN	RNS
	4.48		
(0,2,4,6) ⇒ (0,10,20,30)		1009.17	
(0,3,6,1) ⇒ (0,11,22,25)		115.70	
(0,4,1,5) ⇒ (0,12,17,29)		491.95	
(0,5,2,7) ⇒ (0,13,18,31)		115.70	
(0,7,6,5) ⇒ (0,15,22,29)		115.70.	

4. ARCHITECTURE OF THE NEW JOIN DATABASE COPROCESSOR.

Based on the hashing simulation results and the estimates of speed and cost in chapter 3, the first section of this chapter explains the grounds for choosing the mapping hash method for the hash coder for the HIMOD database computer. In the next section, the connection between the host processor and the join database coprocessor is described, and each of their functionalities in performing the join operation is also illustrated. Then the architecture of the host processor (and the software backend) is described in section 4.3. Next, the architecture of the hardware back end is described with block diagrams. The interface mechanism between the host and the back end is described in the final section.

database

4.1 The Mapping Hash Method as the Choice

After surveying the various hash methods in chapter 3, the mapping hash method is selected for the hash coder in the database coprocessor because it has not only reliable, data independent, and relatively good key distribution, but also it takes only 3 clock cycles in transforming a key to an address if it is implemented in hardware. Those performances in both key distribution and speed of the mapping hash coder are very pursuasive.

In our application environment, the hash coder is the major component in the database coprocessor which is used as a filter device. While long queries are being executed on large database, series of millions and millions of data are waiting to be

This is an inherent problem of the hashed bit array filtering technique. If an actual merge is performed on the passed tuples, the resulting relation may include errors. As explained, the errors are caused by collisions due to hashing.
the host processor has to spend time in discarding those Spurious tuples by tedious key comparisons. The next section shows how HIMOD overcomes this spurious keys problem using the stack oriented filter technique (SOFT).

5.2 Stack Orient Filter Technique (SOFIT) and the New Join Algorithm.

As previously mentioned in chapter 4, the stack orient filter technique (SOFIT) is the main idea used in a new join algorithm. There is a stack containing five items (or elements) which are bit array stone. Initially the lowest bit array store is the item at the top of the stack. The stack pointer (Top) has to keep track of the top of the stack (- the current bit array store), since another bit array store may be added into the top of the stack and the current bit array store may be deleted from the top of the stack. overal primitive operations in the stack data structure such as push, pop, and bottom of stack are provided for use in the new join algorithm. The upper limit in the stack is five since the number of bit array stores is five. Therefore, it is not allowed to push another item when

five items are already stacked up. The operation bottom of stack indicates if the stack pointer (Top) point to the lowest item in the stack. In this situation, the pop operation cannot be applied to the stack because the stack in the SOFT keep at least one bit array store for buckets of a hash table.

go to next page

Every bit array store has N(=50) bits. And if the value of the i-th bit of current and saved B.A.S is '1', the i-th bucket is not an empty bucket, and it's corresponding subset file (or list) contains collected tuples. On the other hand, if the value of the i-th bit of any of them is '0', there has been no key addressed to this bucket by an affiliated hash coder, so no tuple is stored in this bucket.

In the process of the join, there might be three different types of bit array stores (B.A.S) as shown in Figure 5-2; they are Current B.A.S, Saved B.A.S,

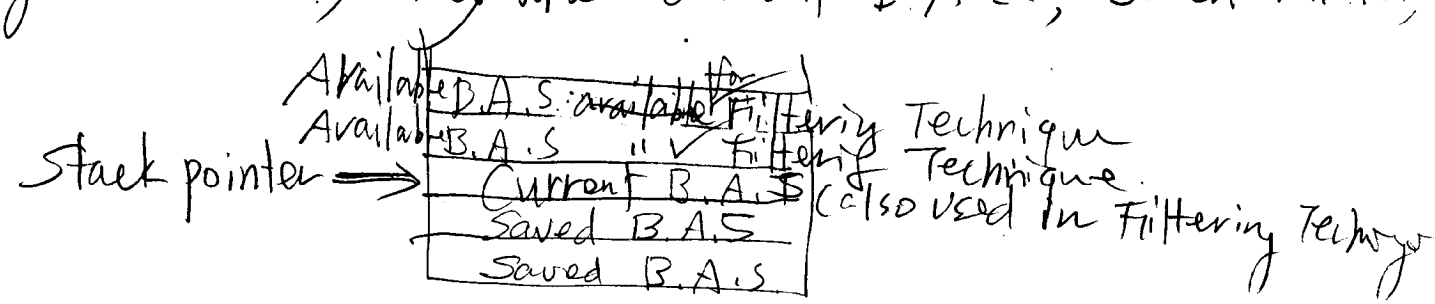


Figure 5-2 Stack Configuration after Two Items are Pushed.

and B A S. which is available for hashed bit array store filtering technique. The current B.A S. is the one that is pointed by the stack pointer, and if is also involved in the hashed bit array store filtering technique. The input tuples are stored in the addressed bucket by the hash coder, which is connected to the current B A S.

→ insert it from the previous page.

Therefore, the first (lowest) B A S has an information for distribution of the whole source and target relations. If both input relations are so large that they cannot fit into the main memory, they are divided into maximum 256 subset files for each relation by the associated hash coder of the B A S. (→ next page) It is also clear that the tuple(s) of the i-th source subset file might have matched tuple(s) in only the i-th target subset file if there any. ⊛ Otherwise, it is obvious that unnecessary (or matchless) tuples can be included in the i-th source and target subset files. The SOFIT is to filter those

① as shown in Figure 5-3. (step 1).
If each BAS produces only one hash address in entire scanning of keys, an output from the five hash address comparators sends a signal to the controller saying that (about all) unnecessary tuples are filtered, as discussed in chapter 4.

(~~since there is approximately one in a trillion chance of having a spurious key (1/(256**5)).~~ (although it is extremely rare)
In this case, all the tuples in the source and target subset files are sent to the host processor for merge.

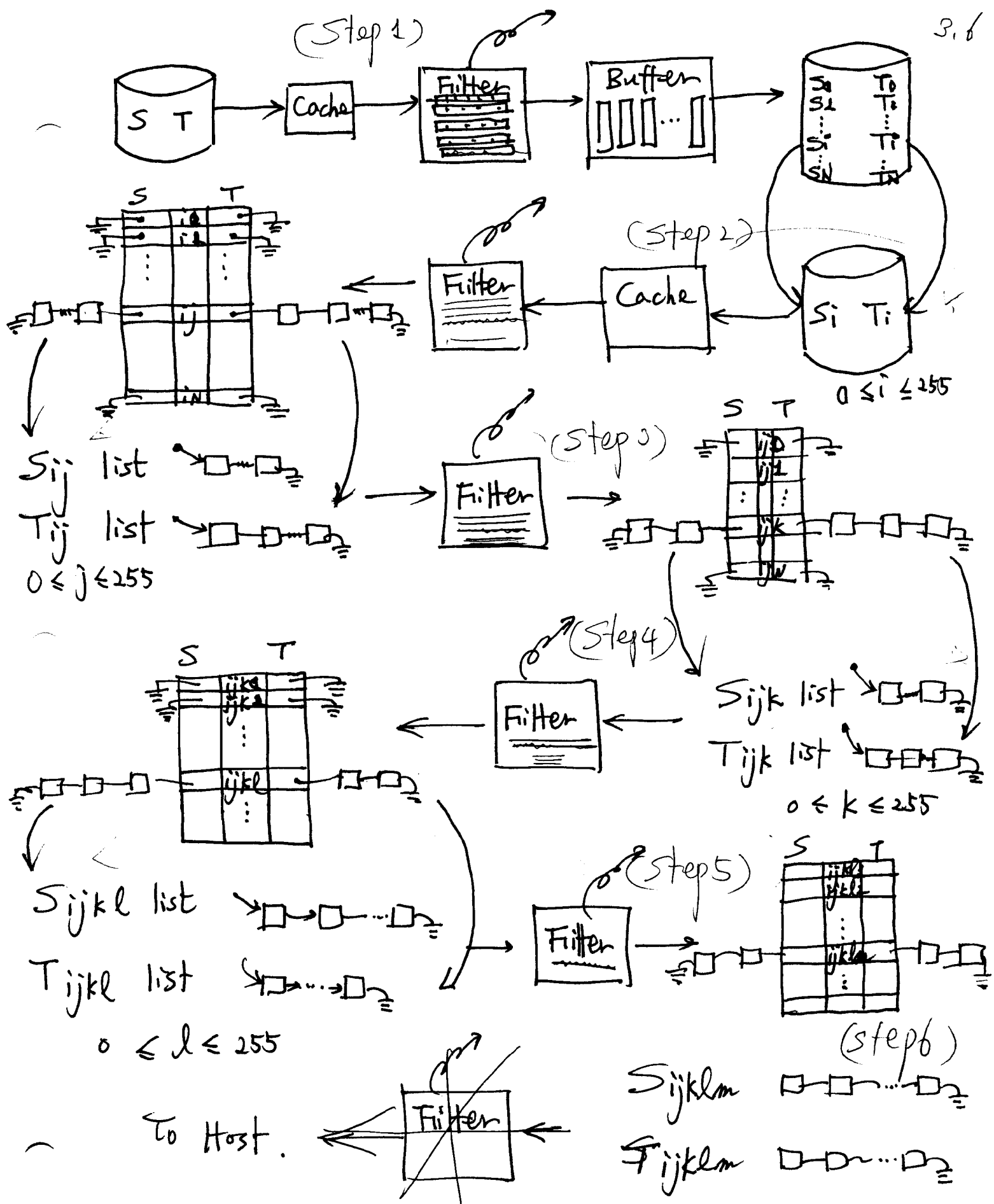


Figure 5-3 The processes in the SOFT.

unnecessary tuples efficiently. Now the i-th ($0 \le i \le N-2$) source and target subset files are to be divided again by the SOFT. However, the next subset files after the i-th one should be saved along with the lowest BAS so that they can be processed subsequently. Thus the current DAS is pushed onto the stack, so the second BAS becomes the current one. Using the three upper BASs and the current BAS, the keys of the i-th source and target subset files are again hashed by the four functionally different hash coders. Those tuples which are passed through hashed bit array filter are stored in the hash-addressed bucket by the hash coder associated with the second (current) BAS.

As shown in step 2 of the Figure 5-3, a hash table which has pointers to either a linked list or a null in each bucket will be created. If each one of the four BASs used in filtering process produces only one hash address in entire scanning of keys, an output from the four hash address comparators sends a signal to the controller indicating that it's o.k. to send the tuples in the only one pair of source and target lists to the host processor for merges, and

the SOFIT goes back to the step 1 to process the next subset file. It quits if no subset file left. When the SOFIT moves to the step 1, internally the current (second) BAS is popped from the stack, and the first (lowest) BAS becomes the current one.

If there is no signal from the four hash address comparators, the ij-th ($0 \le j \le N = 255$) source and target linked lists which has tuples need to be divided again. Again the current BAS is pushed onto the stack, so the third BAS becomes the current one. Using the upper two BASs and the current BAS, the keys of the ij-th source and target linked lists are hashed by the three functionally different hash coders. Those tuples which are passed through the filter are stored the hash-addressed linked list by the third hash coder as shown in step 3 of the Figure 5-3.

So much about the SOFIT has been discussed; therefore, step 4 and step 5 in Figure 5-3 can be explained by the same way. In the step 6, no available BAS is left and unnecessary data have been already filtered,

so the source and target tuples are just sent to the host processor without the filtering process.

Since the SOFT is a recursive procedure, bit array stores are structured as elements of a stack. This nature of recursion in the SOFT simplifies the architectural structure of the DBC.

(1) Initialization
(2) Read in Source and Target Relations from secondary storage
(3) repeat
(4) Clear ~~Bit Arrays in~~ the Current and Upper ~~Part of the Stack~~ BAS(s).
(5) Hash Source Join Attributes Setting Up Bits in Current Bit Array.
(6) Hash Target Tuples and Eliminate Unnecessary tuples by Examining Hash-Addressed Bits in all BAS(s) involved in filtering proces.
(7) ~~Eliminate Unnecessary Source Tuples Examining Each Bits in Current BAS.~~ the ANDed Result of
(8) if Hash Addresses are Identical for All Join Attributes then
Send Headers of Source and Target Tuples Linked Lists.
To the Host Processor for Merge.
(9) if No More Next Hash Address (left in the Current Bit Array) then
(10) if (the current Bit Array is the) Bottom one of the Stack then
(11) Finish Gets True Value
(12) else
(13) Pop the Stack.
(14) if No More Next Hash Address (11) then
(15) If Bottom one of the Stack then
(16) Finish Gets True Value
(17) else
(18) Pop the Stack
(19) if No More Next Hash Addr then
(20) if Bottom One of the Stack then

```
      writeln;
      write(' THE TOTAL NUMBER OF KEYS IN THE TARGET RELATION : ');
      writeln(Target_Count:4);
      writeln;
      write(' THE TOTAL NUMBER OF KEYS IN THE RESULT RELATION : ');
      writeln(Result_Count:4);
      writeln;
      writeln;
      writeln;
      write(' THE TOTAL NUMBER OF HASH CODER USED IN THE JOIN : ');
      writeln(Hash_Count:4);
      writeln;
end;
```

correct.

* Stack_Bottom
* No_More_Next_Bit_Addr
* Save_Next_Bit_Addr

```
{**************** MAIN PROGRAM STARTS HERE ******************}
0
1   begin
2   Initialization;
3   Form_Source_And_Target_Relations;
4   repeat
5     Clear_Current_Upper_Part_Of_Stack;
6     Hash_Source_And_Target_Relations;
7     if Only_One_Bit_Set_After_Hash(Hash_Value) then
      begin
8       Merge_Relations_And_Print_Out(Hash_Value);
9       if No_More_Next_Addr then
10      begin
11        if Bottom_Of_Stack then
12          finish := true
13        else
14        begin
15          pop;
16          if No_More_Next_Addr then
17          begin
18            if Bottom_Of_Stack then
19              finish := true
20            else
21            begin
22              pop;
23              if No_More_Next_Addr then
24              begin
25                if Bottom_Of_Stack then
```

```
                    finish := true
                else
                begin
                  pop;
                  if No_More_Next_Addr then
                  begin
                    if Bottom_Of_Stack then
                      finish := true
                    else
                    begin
                      pop;
                      if No_More_Next_Addr then
                      begin
                        if Bottom_Of_Stack then
                          finish := true
                        else
                        begin
                          Assign_Source_And_Target;
                          Save_Next_Addr_Bit;
                          push;
                        end;
                      end;
                    end
                    else
                    begin
                      Assign_Source_And_Target;
                      Save_Next_Addr_Bit;
                      push;
                    end;
                  end;
                end
                else
                begin
                  Assign_Source_And_Target;
                  Save_Next_Addr_Bit;
                  push;
                end;
              end;
            end
            else
            begin
              Assign_Source_And_Target;
              Save_Next_Addr_Bit;
              push;
            end;
          end;
        end
        else
        begin
          Assign_Source_And_Target;
          Save_Next_Addr_Bit;
          push;
        end;
      end;
    end
    else
    begin
      Assign_Source_And_Target;
      Save_Next_Addr_Bit;
      push;
    end;
  end
end
```

```
  else
  begin
    Assign_Source_And_Target;
    Save_Next_Addr_Bit;
    push;
  end
until finish;

Print_Statistics;

end.
//GO.SYSIN DD *
ABCDEFGHIJKLMNOPQRST
UVWXYZ01234 abcdefgh
ijklmnopqrstuvwxyz56
789#.,'-_$
65 66 67 68 69 70 71 72 73 74
75 76 77 78 79 80 81 82 83 84
85 86 87 88 89 90 48 49 50 51
52 32 97 98 99 100 101 102 103 104
105 106 107 108 109 110 111 112 113 114
115 116 117 118 119 120 121 122 53 54
55 56 57 35 46 44 39 45 95 36
2729 2063 7927 5087 3583 1307 7687 1523 3643  223
 103  523 7129 5669 3229 7789 8527 4969 2549 1721
3469 5189 5563 5981 4021 3187 3167 4409 6827 1109
 461 6323  769 4363 4801 1481 6367 7963 1747 2203
5081 3083 6547 3727 7069 2887 2221 8009 1987 2161
2683 4583 4127 7541 6361  967 5627 2309 4787 6581
8287  743 5347 3709 6763 1021 1949 5449 3041 3907

2063 7927 5087 3583 1307 7687 1523 3643  223 7481
 523 7129 5669 3229 7789 8527 4969 2549 1721 3929
5189 5563 5981 4021 3187 3167 4409 6827 1109 4241
6323  769 4363 4801 1481 6367 7963 1747 2203 1061
3083 6547 3727 7069 2887 2221 8009 1987 2161 3301
4583 4127 7541 6361  967 5627 2309 4787 6581  641
 743 5347 3709 6763 1021 1949 5449 3041 3907 6689

7927 5087 3583 1307 7687 1523 3643  223 7481 5483
7129 5669 3229 7789 8527 4969 2549 1721 3929 1607
5563 5981 4021 3187 3167 4409 6827 1109 4241 1123
 769 4363 4801 1481 6367 7963 1747 2203 1061 3823
6547 3727 7069 2887 2221 8009 1987 2161 3301 8167
4127 7541 6361  967 5627 2309 4787 6581  641  443
5347 3709 6763 1021 1949 5449 3041 3907 6689 6067

5087 3583 1307 7687 1523 3643  223 7481 5483  401
5669 3229 7789 8527 4969 2549 1721 3929 1607 6121
5981 4021 3187 3167 4409 6827 1109 4241 1123 4129
4363 4801 1481 6367 7963 1747 2203 1061 3823 6949
3727 7069 2887 2221 8009 1987 2161 3301 8167 3449
7541 6361  967 5627 2309 4787 6581  641  443 7349
3709 6763 1021 1949 5449 3041 3907 6689 6067 5521
```

5.3. The New Join Algorithm

The new join algorithm is designed based on the stack oriented filter technique (SOFT), so the fundamental data structure used in this algorithm is a stack as explained in the previous section. The push and pop are the procedures to operate on the stack; push inserts a bit array store (BAS) onto the top of the stack and pop deletes one from the top of the stack. The stack pointer always points to the current BAS (the item at the top of the stack), by incrementing it's value when push is called and by decrementing it's value when pop is called. By refering to the value in the stack pointer, the function Bottom_Of_Stack can tell whether the stack pointer points to the first (or lowest) BAS as the current item of the stack.

In the new join algorithm, as shown in Figure 5-4, there are several other frequently used procedures such as ~~Assign_Source_And_Target~~, Save_Next_Bit_Addr, and No_More_Next_Bit_Addr. The procedure Assign_Source_And_Target gets

the header pointers of both source and target linked list, based on the saved next address of the current BAS, and the tuples in the linked lists are to be processed through the filter again. As explained in chapter 4, each BAS in DBCP has an associated address register, and next bit address is saved in that register and incremented to keep track of another next bit address. Whenever the procedure Assign_Source_And_Target is called, another next bit address which has the value '1' in the current BAS is found and saved for the next process with that BAS by the procedure Save_Next_Bit_Addr. Then the procedure push saves the contents of the current BAS and increments the stack pointer so that the next upper BAS becomes the current BAS (or top of the stack).

When the procedure pop is called, the stack pointer gets decremented so that the BAS directly under the current BAS becomes the current BAS. After the pop, the boolean function No_More_Next_Bit_Addr is always called to see if there is any saved next bit address in the current BAS. If none, the current BAS is checked if

if is the first (lowest) BAS. If so, the join process is terminated by breaking the repeat loop.

The new join algorithm has been simulated, and the simulation program in Pascal is appended in Appendix I. The algorithm shown in Figure 5-4, is an explanatory version of the main module of the simulation program. After an appropriate initialization (2) process (including finish := false), the algorithm reads in the two input files for source relation and target relation from the secondary storage (3). Then there is a large repeat loop (4) starting with clearing the current and upper BAS(s) for use in hashed bit array filtering technique (5). Next, the algorithm scans the tuples in the source linked list (or file) hashing their join attributes and setting up hash-address bits in the corresponding BAS(6). The source tuples are divided by the hash coder of the current BAS, and stored in the addressed bucket (linked list). Then it hashs the join attributes of the target tuples in the list, and it detects and eliminates unnecessary tuples by examining the ANDed result of the all BAS(s) involved in hash addressed bit array filtering (6). The target tuples which are not filtered in this step (6) are

saved in the buckets which are addressed by the hash coder associated with the current BAS. 4

Then there is a if-statement (7) testing if the produced hash addresses are all identical by either examining if only one bit is set in all BAS involved in filtering process or examining the output signal from the hash address comparators as in HIMOD. If so, it sends headers of source and target linked lists (or subset files) for merge (8), and it checks if there is any saved next bit address (9). If there is next bit address, it assigns headers of source and target linked list in the next bit address ~~bucket to currently~~ used temporary variables for filtering process (15). Then the next bit address (which is in next address register in HIMOD) is incremented until another bit address which has '1' in its content in the curren BAS is found. (16) If no such next bit address is found, it assigns null value to the next bit address. After that, it pushes the current BAS onto the stack, goes back to the beginning step of the repeat loop starting with clearing the current and available BASs (5), and hashes the assigned source and target relations in the step 15 (6).

In step 9, if there is no more next bit address then it checks if the current BAS is, the first (lowest) BAS (11). If so, it assigns true value to the repeat condition variable "finish" (12) and the join process is terminated. If there is saved BAS(s), then it pops the BAS (15), and checks if there is saved next bit address (16). If there is next bit address, then it does the same sequence as explained in the steps 75, 76, 77 (61, 68, 69) using the next bit address as the bucket address of the current BAS. If there is no next bit address, it does either pop the stack (22) or assigns true to the variable finish (19) based on the boolean output of the Stack-Bottom (18).

There are several nested if-then-else statements which include the same code patterns. Accordingly, this algorithm also can be implemented using a recursive routine. Since the stack can have only five items in maximum, this nonrecursive algorithm is good enough to be implemented easily in the controller (ROM) of the DBCP.

CHAPTER 5

A NEW HASH BASED JOIN ALGORITHM.

⇒ There are other relational database operations which have hashing in their natures, and those relational operations are projection, union, difference, and intersection. The last section discusses how these operations also can utilize the hash coder in the DBCP effectively.

5. A NEW HASH-BASED JOIN ALGORITHM.

In this chapter, the new hash-based join algorithm using stack oriented filter technique (SOFIT) will be introduced. The stack oriented filter technique supports divide and conquer strategy in filtering unnecessary data. The new hash-based join algorithm and the SOFIT are explicitly explained in section 5.1. The simulation results of the new join algorithm are shown and compared with the performance of the conventional join in section 5.2. The final section compares the execution times of both when the new hash-based join algorithm is run on the software back end and when it is run on the hardware back end.

5.1. Stack Oriented Filter Technique (SOFT) and the New Join Algorithm.

5.1.1 Bit Vector Filtering Technique.

The new database computer HIMOD uses bit vector (or hashed bit array store) filtering technique <BABB1, VALD 1>, since it showed dramatic improvement in all join algorithms without substantially increasing hardware cost <DEWI3, GADA2, SHAP1, VALD2, SCHN1>.

This section describes how the bit vector filtering technique is applied to the join method of

5.5.1 Project. (Eliminating Duplicates)

After the project operation selects from each tuple in the input relation only those attributes that specified in the attribute list, there might be duplicate tuples in the resulting relation

For example, in Figure, there are a relation R, and a resulting relation for PROJECT R (B, D).

Relation R

A	B	C	D
d	e	f	k
b	d	g	h
e	c	i	m
a	d	i	h

PROJECT R (B, D)

B	D
e	k
d	h
c	m
d	h

Figure 6-1

intermittent

In the resulting Relation, the second and the fourth tuples have the same values (d and h) for the attributes B and D, and duplicate tuples like the fourth tuple has to be eliminated.

DBMSs generally sort the tuples in the intermittent resulting relation in alphabetical order, and search and eliminate unnecessary tuples. To find duplicate tuples in the sorted relation, two pointers are usually used to keep track of the tuples being compared. The values in the attributes of the tuple pointed by the first pointer are compared with those of the next tuple pointed by the second pointer. During the comparison process, both pointers are incremented whenever necessary until they reaches the last tuple in the sorted relation.

~~In HIMOD, hashing is used instead of sorting for the project operation taking advantage of the fast hash coder built in the DBCP. The values (or combined value) of attributes in the attribute list are hashed and the hash-addressed bucket is searched for a match. if tuples already have been stored.~~

In HIMOD, hashing is used instead of sorting for the project operation taking advantage of the fast five hash coders in the DBCP. The partial value (or a partial combined value) of attributes in the attribute list is hashed by the five statistically independent hash functions, and using the five hash addresses, the corresponding bit array stores (BAS's) are examined. If at least one bit in any BAS is not set, no identical tuple already has passed through the filter so that the input tuple is immediately stored in the hash table based on the hash address produced by the first hash coder without comparing the tuple with previously stored tuples in the hash-addressed bucket. Then the corresponding bits in the five BASs are marked by the five produced hash addresses.

If all of the hash addressed bits in the five BASs are set, it is probable to have an identical tuple which has already passed through the filter. Therefore,

the whole input tuple has to be compared with the tuples in the hash-addressed bucket indicated by the first hash coder. If there is a match, the tuple is instantly eliminated and next tuple is brought into the hash coder for the same process. If there is no match, the tuple is stored in the bucket. Finally the tuples stored in the hash table are included in the resulting relation for the project operation.

The advantage in using the DBCP is to figure out the necessity of tuple comparisons at the beginning by examining the bits in the five BASs. As a result, the time for comparisons might be saved in some extent.

~~If there is a matched tuple, one of them is not necessary and eliminated. Otherwise, the tuple is stored in the hash-addressed bucket. This process repeats until no tuple is left in the intermittent resulting relation. The tuples which are accumulated in the hash table will consist of the final resulting relation.~~

When the intermittent resulting relation does not fit in the real memory, the relation should be divided based on the hash address produced by the first hash coder.
Depending on how the main memory is used (e.g., a hash table, output buffers, and both output buffers and a hash table), the project method can be different like the hash-based join methods discussed in chapter 2. (Simple, GRACE, and Hybrid hash joins)
Once the intermittent resulting relation is divided into subset files, tuples in a subset file are

hashed out through that filter. So the speed in hash address calculation is crucially important factor in selecting a hash method. If one is looking for super fast hardware hash coder, he will consider the digit analysis and folding hash methods among the current well known methods. But the digit analysis method is for static inputs, so it is excluded in this type of application. As discussed, the folding method doesn't seem very reliable. After Lum reviewed major hash methods, he concluded that the digit analysis and the folding hash methods are erratic. To make sure that the distribution performance is good, (1 see next page) it's better to combine the folding technique with other. As Maurer mentioned, an improved solution to a specific problem may almost always be achieved by using a combination of techniques <MAVR1>. The mapping

① And also Buckholz said that it should be noted that folding is not as good as multiple-precision division based on his experiment

---

hash method is the combination of the mapping (converting each character in a key to a prime number) technique and the folding technique.

This mapping hashing technique is designed not only for good distribution, but also for fast address calculation with possible hardware aids. The parallel processing in both transforming each character into a prime number and calculating each bit value in a hash address by means of hardware leads to producing a hash address within 3 clock cycles. Other hash methods cannot take advantage of parallel processing effectively because of their algorithmic natures in hash address calculation.

And some of the well known hash functions such as the Midsquare and the Fold-boundary show data dependency problem recognizably, and other hash functions like the multiplicative, the radix, and the random show a sign that they can perform poor for some data set.

The mapping hash coder in hardware is relatively inexpensive comparing other hardware hash coders. It requires 120 gates and sixteen 64x16 bits RAMs since the mapping hash function does not use complex mathematical operations like multiplication and division as other methods such as midsquare, multiplicative, radix, random, and algebraic coding use.

Another requirement of a hash function for this specific application is that the database coprocessor has to have 5 statistically independent hash coders so that by 5 functionally different hash methods, the 5 hash addresses should be produced within the same period of time and they must be totally irrelevant each other. The reason for this requirement will be explained in the next section which describes the filtering technique using the five functionally different hash coders. For this particular requirement, the mapping hash method is advantageous since based on the stored contents (selected prime numbers) of the RAM, each hash coder calculates a hash address in it's unique way. Because each of the 5 RAMs has a different set of prime numbers, the five hash addresses generated at a time are irrelevant. And the address calculation time for each hash coder is always the same. This characteristic of

statistical independence is an asset of the mapping hash function. Not every hash function has such a property. To provide each hash coder with statistical independence, the modification of each hash function is often unnatural or brute-force so that it is hard to claim that each hash function is 100 percent functionally different. In addition to that problem, it's also hard to maintain the same address address calculation time for each modified hash coder.

Consequently, it is concluded that the mapping hash coder in hardware satisfy the previously mentioned three requirements (relatively good distribution, extremely fast, and inexpensive) and provide the property of statistical independence in each hash function. Therefore, if it is adapted for the database hash coder in the database computer HIMOD.

4.2 An Overview of HIMOD Architecture

As mentioned previously, the primary goal is to implement a database computer that supports frequently used and time-consuming software database management functions such as the join operation using a database coprocessor. The main approach is to maximize the filtering effect in the join process, so the database coprocessor is used as a filter device. The unnecessary data are filtered by the filter as soon they are detected and the only tuples that will be included in the resulting relation will be transmitted to the host processor. The transmitted source tuple(s) and target tuple(s) are merged together by the host processor. Therefore, the parallelism is exploited in the join operation so that the filtering process and the merging process

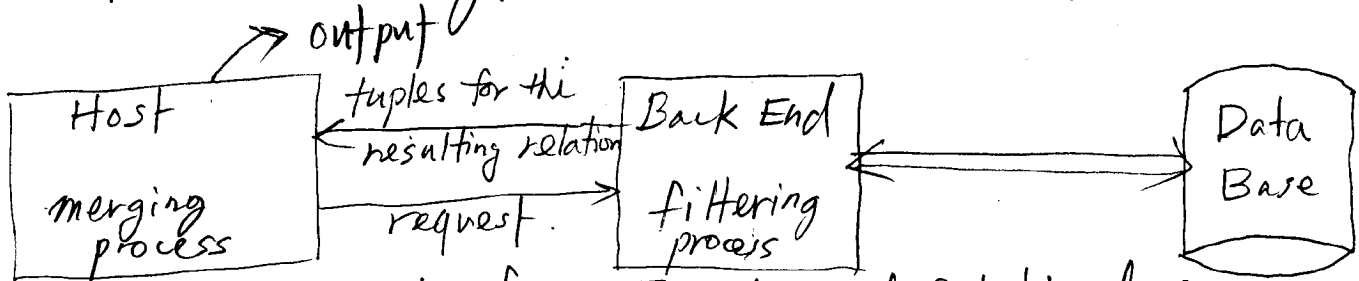


Figure 4.1 Example of the Execution of Relational Join.

are concurrently executed by the back end and the host respectively. The main idea is that the tuples which the host processor receives are the necessary ones in producing a resulting relation, and they are only a handful in most cases, so the host processor is not burdened with carrying many join attributes and comparing them for a match. This idea is successfully accomplished by stack oriented filter technique and the new hash-based join algorithm which will be explicitly explained in the next chapter.

HIMOD uses a Motorola 68030 32-bit microprocessor as the host processor, the back end processor communicates with the host processor through a protocol, defined as the M68000 coprocessor interface <MOTO1>. A back end processor adds additional database instructions,

and additional registers and data types to the programming model that are not directly supported by the host processor architecture. The necessary iteractions between the host processor and the database coprocessor that provide a database operation are transparent to the programmer. The programmer, therefore, does not need to know the specific communication protocol between the host processor and the database coprocessor because this protocol is implemented in hardware. Thus, the database coprocessor can provide capabilities to the user without appearing separate from the host processor

The connection between the host processor unit (HPU) and the database coprocessor (DBCP) is 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's function codes and address bus) selects the DBCP. The host processor and the coprocessor configuration is shown in figure 4.2.

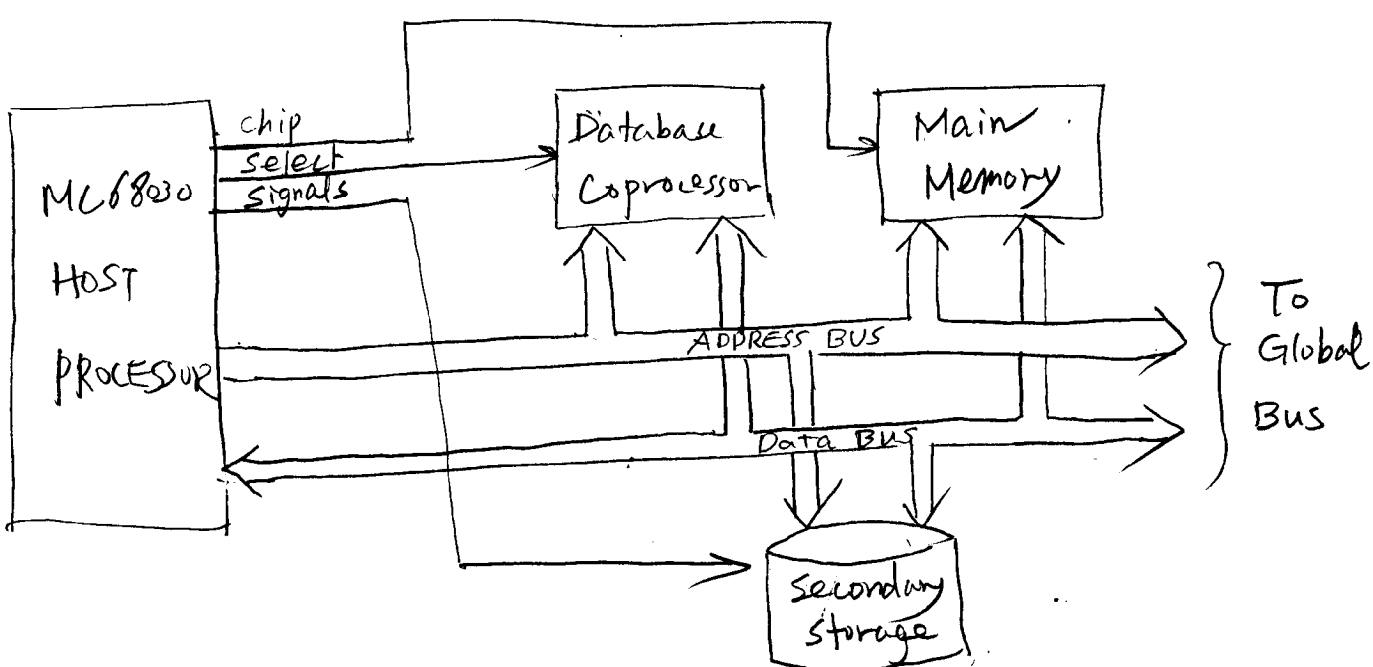


Figure 4-2. Coprocessor Configuration

All communications between the HPU and the DBCP are performed with standard M68000 Family bus transfers. The DBCP is designed to operate on 32-bit data bus. The DBCP contains a number of coprocessor interface registers, which are addressed by the host processor in the same manner as memory.

4.3 Architecture of the Host Processor

Since the MC68030 is selected for the host processor of HIMOD, the general description and features of the MC68030 <MOTOR> are briefly illustrated in this section. The MC68030 is a second-generation full 32-bit enhanced microprocessor from Motorola. The MC68030 combines a central processing unit (CPU) core, a data cache, an instruction cache, an enhanced bus controller, and a memory management unit in a single VLSI device. This processor operates at clock speeds beyond 20 MHz. And it is implemented with 32-bit registers and data paths, 32-bit addresses, a rich instruction set, and versatile addressing modes.

The MC68030 provides the non-multiplexed bus structure with 32 bits of address and 32 bits of data. The MC68030 bus has an controller that supports both asynchronous and synchronous bus cycles and burst data transfers. It also supports a dynamic bus

. sizing mechanism that automatically determines device port sizes on a cycle-by-cycle basis as the processor transfers operands to or from external devices.

A block diagram of the MC68030 is shown in Figure 4.3. The instructions and data required by the processor are supplied from the internal caches whenever possible. The memory management unit (MMU) translates the logical address generated by the processor into a physical address using its address translation cache (ATC). The bus controller manages the transfer of data between the CPU and memory or devices at the physical address.

Figure 4.3 MC68030 Block Diagram

As illustrated in the MC68030 User's manual <MOT01>, the features of the MC68030 microprocessor are:

1. Object code compatible with the MC68020 and earlier M68000 microprocessors
2. Complete 32-bit non-multiplexed address and data buses
3. Sixteen 32-bit general purpose data and address registers
4. ⋮
14. Processor speeds beyond 20 MHz

The prime reason behind choosing the MC68030 is that it has a 4-gigabyte logical and physical addressing range, which is sufficient for most of the database management systems. Another motive is that the simple M68000 coprocessor interface incorporates the design of the database coprocessor.

HIMOD may use MC68030 as a software back end. This software back end is dedicated to perform database management functions only, so the contents of microsequencer and control store should be entirely replaced with the microcodes of database operations. Therefore, there is no hardware modification or enhancement on the database coprocessor except interface unit, but innovative software architecture is implemented on the back end.

4.4 Architecture of the Hardware Back End

The new hardware back end processor is intended primarily for use as a database coprocessor (DBCP) to the MC68030 32-bit microprocessor unit (HPU). This database coprocessor provides a high performance filter unit. The major features of the DBCP are:

1. Fully concurrent instruction execution with the main processor.
2. Five fast hash coders produce five statistically independent hash addresses simultaneously within 3 clock cycles.
4. Five hash address comparators which are attached to the corresponding hash coders tells whether only one kind of keys (or join attributes) has passed through the filter or not.
3. Single-bit wide (256 bits) RAM is connected to each hash coder to keep the records of generated hash addresses.

As shown in Figure 4-5, the database coprocessor (DBCP) 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 send control signals to the bit array filtering unit to execute the intended database operation.

For both the bus interface unit and the processor control unit, the DBCP uses the conventions of MC68881 and MC68882 floating-point coprocessor chips <MOTO2>

4.4.1 Bus Interface Unit

The bus interface unit contains the coprocessor interface registers (CIRs), the CIR register select and DSACK timing control logic, and the status flags which is used to monitor the status of communications with the host processor as shown in figure 4-6. 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

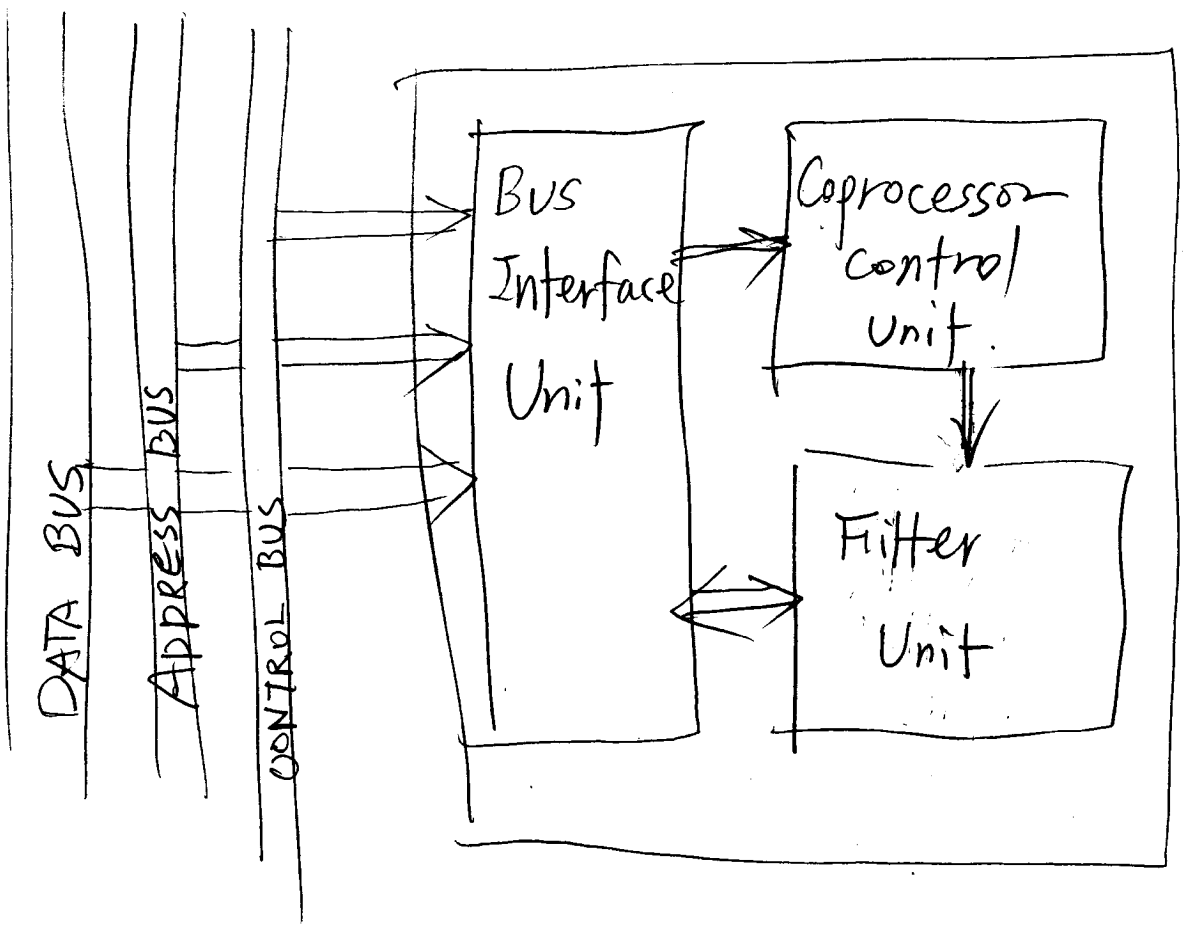


Figure 4-5. DBCP Simplified Block Diagram

standard M68000 Family bus transfers <MOTO1>. The M68000 Family coprocessor interface is implemented as a protocol of bus cycles in which the host processor reads and writes to these CIRs. The MC68030 host processor implements this general purpose coprocessor interface protocol in hardware and microcode.

When the host processor detects a DBCP instruction (e.g. join operation), the host processor writes the command word of the instruction to the memory-mapped command CIR, and reads the response CIR. In this response, the bus interface unit encodes requests for service required of the host processor to support the DBCP in performing the intended database operation. After the host processor serves the DBCP request(s), the host processor can fetch and execute subsequent instructions.

The coprocessor interface should allow concurrent instruction execution, so the synchronization during

host processor and DBCP communication should be accomplished. Concurrent or nonconcurrent instruction execution is determined based on nature of each coprocessor instruction. While the execution of most DBCP instructions can be overlapped with the execution of host processor instructions, concurrency is completely transparent to the programmer. Therefore, from the programmers' view, the host processor and DBCP appear to be integrated onto a single chip.

However, the DBCP does not need to run at the same clock speed as the host processor because the bus is asynchronous. Due to this aspect, the database management system performance can be customized. And since the M68000 Family coprocessor interface also permits coprocessors to be bus masters, the DBCP functions as one. The DBCP can fetch all tuples or attributes and store all results. In this manner, the DBCP, as a filter device, effectively perform intended database operation by filtering unnecessary data. This type of coprocessors is refered to as DMA coprocessors.

The DBCP DMA coprocessor operates as bus slave while communicating with the host processor across the coprocessor interface, but also have the ability to operate as bus masters and directly control the system bus. The second type based on bus interface capability is called non-DMA coprocessors that always operate as bus-slaves. To speed up the data transfers between memory and the DBCP, the DBCP requires a relatively high amount of bus bandwidth, so it needs to be implemented as a DMA coprocessor. Therefore, DBCP provides all control, address, and data signals necessary to request and obtain the bus, and then performs DMA transfers using the bus.

4.4.2 Coprocessor Control Unit

The control unit of the DBCP contains the clock generator, a two-level microcoded sequencer, and the microcode ROM. The microsequencer is either executing microinstructions or awaiting completion of accesses that are necessary in continuation of executing microcode. The microsequencer sometimes controls the bus controller which is responsible for all bus activity. And the microsequencer also controls instruction execution and internal processor operations such as setting of condition codes and calculation of effective addresses. The microsequencer provides the macroinstruction decode logic (instruction decode register and instruction decode PLA) and 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 controls the parallel operation of the functionally equivalent slices of the filter unit.

4.4.3 Filter Unit

One of the main tasks of DBCP is to release the host from tedious database manipulation for the join by filtering the tuples that do not have any potential for inclusion in the resulting relation and sending only the potential tuples to the host processor. The filter unit is the heart of the coprocessor in determining unnecessary data and discarding them.
There are several major approaches in implementing a filter device <BANC1, BABB1, HSIA1>, but their fundamental concepts are the same such that those filters trap irrelevant data when data transferred between the secondary storage device and the main memory.

~~HIMOD database machine adapts hashed address bit array stores filtering technique which is introduced in CAFIS (Content Addressable File Store) database machine <BABB1>.~~ As shown in Figure 4-7, the filter unit of DBCP includes an address filter, five functionally different

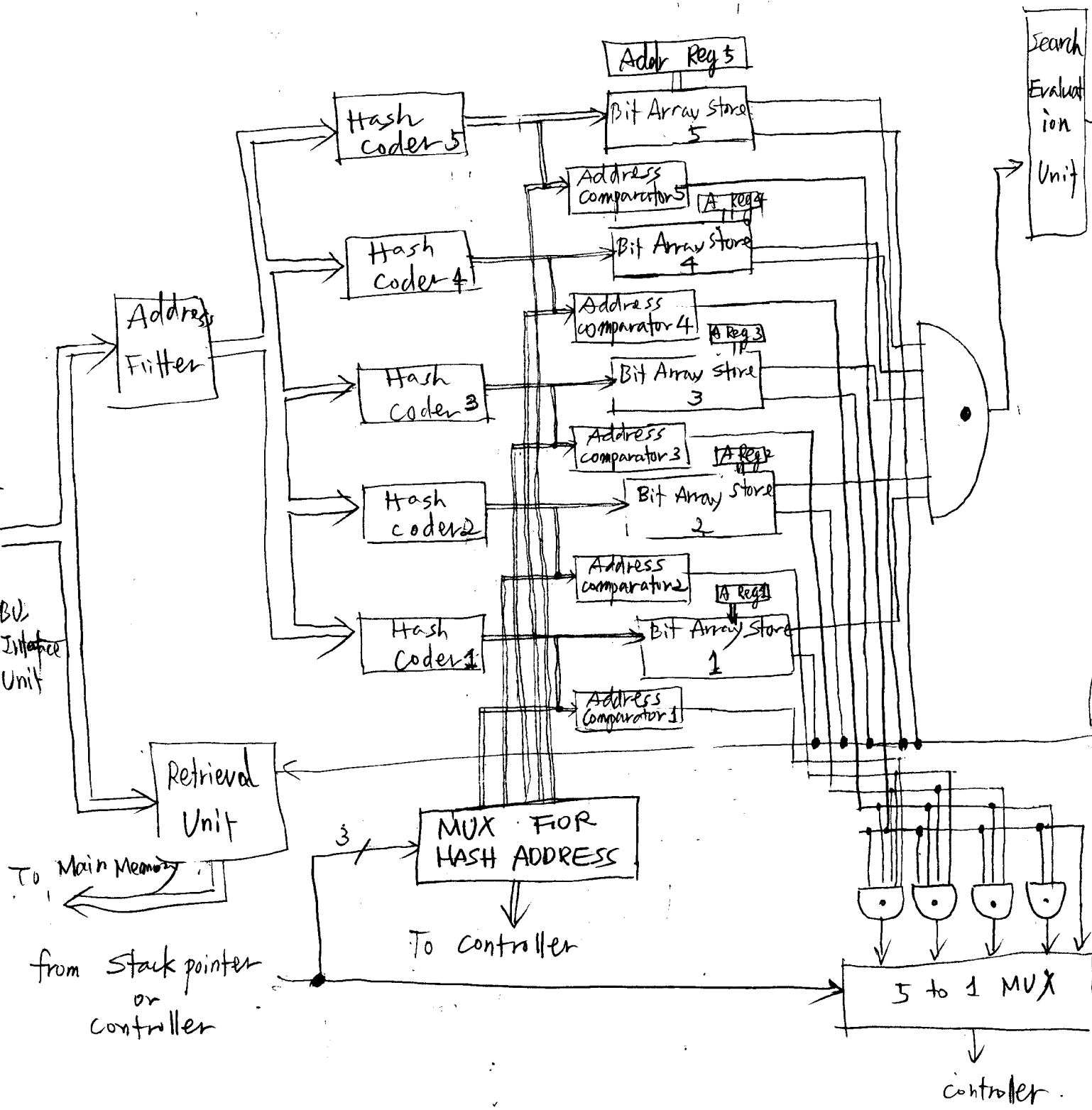


Figure 4-7.

DDCP Architecture

mapping hash coders with associated bit array store, address register and address comparator. As previously mentioned, the bit array store is a single-bit wide random access memory (256 bits RAM), and each bit in the bit array store is addressed by a hash address. The address comparator consists of a register to keep a record of the first hash address produced by the corresponding hash coder and a bunch of exclusive-OR gates, a OR gate and a JK flip-flop to compare an incoming hash address with the first produced hash address as shown in Figure 4-8. For the first hash address loading, a flip-flop and an AND gate are needed to control the load input for the address register. First of all, the flip-flop should be cleared before hashing join attributes in an input list or file, so the output of flip-flop has initially zero value and the inverse of the output is '1'. When a start signal from microcode ROM gets '1', the output of the AND gate is '1' since both inputs of the gate are '1'. This output of the AND gate trigers the flip-flop to send a

see next page 9.1 ~9.6

or a saved next bucket address.

→ goto 10

(2) Each bit array store is associated with a address register which is equipped with an incremant function so that the address register keeps track of the next bucket address to be processed and provides it to the connected bit array store. Therefore, each bit array store has a built-in multiplexor to select right address at any time. The controller sends signals to the control lines of the multiplexor for right selection of an address.

HIMOD database computer adapts hashed address bit array filtering technique which is introduced in CAFS (Content Addressable File Store) database machine <BABB1>, since this filtering technique demonstrated dramatic improvement in all join algorithms without substantially increasing hardware cost <DEWI3, QADA2, SHAP1, VALD2, SCHN1>.

Even if there is explicit detail about the hashed address bit array filtering technique in chapter 5, it is necessary to explain briefly how HIMOD filters unwanted data in performing the join operation. The HIMOD database computer reads the tuples in the source relation from a file (or a linked list), and each value of the join attribute is transformed into five hash addresses by five functionally different hash coders (that is, at this point, the current stack level is at the lowest). To make the explanation simple, it is assumed. These five addresses are used to mark the corresponding bit array store. Then the machine reads the target relation, and the five hash coders again hash each value of the join attribute. Using the five hash addresses,

HIMOD verifies if the hash-addressed bits have already been set. If all five bits have been set, the join attributes of the target relation may match with those of the source relation so that those matched tuples are further processed through the DBCP filter if necessary; otherwise, sent to the host computer to produce the tuples of the resulting relation. The unwanted tuples are detected and discarded by the DBCP since they will not be included in the resulting relation.

If the join condition attribute is called key K, its hashed value will be represented as H(K) where H is a hash function. The mapping hash method assures that each mapping hash coder in the DBCP generates a statistically independent hash address. The hash addresses are represented as M1(H1(K)), M1(H2(K)), M3(H3(K)),

M4(H4(K)), and M5(H5(K)), where H1(K), H2(K), H3(K), H4(K), and H5(K) are different hashed values, and M1, M2, M3, M4, and M5 are corresponding bit array stores. The hash addresses set the corresponding bits to '1' in the bit array stores. After scanning keys and setting up bits in the bit array stores, the output of these five bit array stores are logically ANDed when key K is read from the bit array stores. If the output of the AND gate is '1', then key K is thought to be in the bit array stores. Otherwise, key K has not been hashed to set the bit array stores.

In summary:

To write key K to the bit array stores:

M1(H1(K)) := 1, M2(H2(K)) := 1, M3(H3(K)) := 1,
M4(H4(K)) := 1, M5(H5(K)) := 1.

To read key K from the bit array stores:

M1(H1(K)) = 1 & M2(H2(K)) = 1, M3(H3(K)) = 1.
M4(H4(K)) = 1 & M5(H5(K)).

The above discussion shows how the hashed address bit array stores filtering technique works in HIMOD database computer. Henceforth the hardware structure to enhance this filtering technique will be explained. The architecture of the DBCP can be characterized by a stack oriented structure of the five bit array stores. If there is any bit array stores that is lower than the current bit array store, they are saved in the stack and deactivated in the filtering process. The current and higher than the current bit array stores are participated in the filtering process. The contents of participating bit array stores are cleared first and the bits in the bit array stores are marked as the hash addresses are produced. When a bit array store is saved in the stack, a file or a list of input tuples are divided and distributed into the buckets in the hash table by the hash coder in the prior level of the stack.

Therefore, the divided list of source tuples and that of target tuples are passed through the filter again using the current and higher bit array stores if their join condition attributes are not detected to be identical. Therefore, 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. So the ultimately partitioned lists of the source and target tuples are sent to the host processor for final screening and merge to produce resulting tuples.

To efficiently determine whether the scanned source tuples and target tuples have the same join attribute or not, an hash address comparator is attached to each of the five hash coders. From now on, how the hash address comparator (or address comparator) is designed and how it sends a signal to stop dividing the tuples.

signal ('1') to load the first produced hash address. Once the first address is loaded, the flip-flop is cleared so that the hash addresses hereafter will not be loaded into the address register.

As many exclusive-OR gates as the number of bits in a hash address (or address register) are needed in each hash address comparator. The first bit of the address loaded in the address register and that of incoming hash address are inputed to 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 such that one bit is '1' and another is '0', the output is '1', and it is passed to the OR gate. This OR gate receives all the resulting output signals from those exclusive-OR gates simultaneously. If all of the resulting bits are '0', the output of the OR gate is '0' saying that both hash addresses are identical.

If at least one of the resulting bits from the exclusive-OR gates is '1', the output of the OR gate becomes '1' saying that the loaded hash address in the address register and the incoming hash address are different. see next page ② Therefore, the five structually identical hash address comparators in the DBCP generate output signal at the same time. see next page ①

As shown in Figure 4-9, The five hash address comparators are stacked up. Based on the value in the stack pointer register (or the value from the controller), the 5 to 1 multiplexer selects one from the five inputs. When the stack pointer points to the first (lowest) stack level, all the outputs from the address comparators are ANDed together, and the resulting output of the AND gate (A) is selected by the multiplexer. If the stack pointer points to the second stack level, the first (lowest) bit array store is stored (saved) in the stack and is not written until

for address comparison. 11,5

① Hereafter, the hardware which is required to detect whether all the join attributes in a file or list are identical or not is explained. The purpose of this hardware is to inform the controller whether the input file or list has to be divided further or not so that the DBCP eventually sends the source and target tuples which has the same join attribute to the host processor for concatenation(s) of those source and target tuples.

Then

② The output ('1') of OR gate trigers the K input of the JK flip flop which is initially cleared to to have '1' in output, so the output of the JK flip-flop becomes '0'

the controller send memory write signal to the bit array store. Therefore, the output of the first address comparator is excluded from the inputs into the AND gate (B), and outputs of the second, third, fourth, and fifth hash address comparators are ANDed together. The output of the AND gate (B) is again selected by the multiplexer since the controller, based on the current stack level, tells the multiplexer to select the output. By the same way, if the indicated stack level is the third level, the first and second bit array stores are saved in the stack. and the multiplexer selects the output of the AND gate (C) which receives outputs from the third, fourth, and fifth address comparators as inputs. And if the indicated stack level is the fourth level, the first, second, and third bit array stores are saved in the stack and the multiplexer chooses the output of the AND gate (D) which receives outputs from the fourth and fifth address comparators as inputs.

If the indicated stack level is the fifth (highest) level, the four lower level bit array stores are saved in the stack and the multiplexer selects the output directly from the fifth level address comparator.

The single bit output from the multiplexer trigers the attached JK flip-flop if all of the hash address comparators which are equal or higher than the current stack level tells that only one kind of hash address has been produced from each hash coder after a whole input file or list has been scanned. The output value of the JK flip-flop is sent to the controller for either further division process or conquer process (... sending the list of the tuples left after the filtering process to the host processor for a merge). (next page) ①

The whole filter unit is designed to support the divide and conquer strategy in performing the

join relational database operation. The major concern in the full divide and conquer strategy in the join is to know when no further division of input is necessary. The address comparators determines whether the scanned tuples have the same join attribute or not and informs the controller for either further dividing process or sending those wanted tuples to the host processor.

---

go to previous page

① Even though the output signal says that no further division process is necessary, there is approximately one in a trillion chance (1/256**5) to pass an unwanted key. The final screening (direct comparison) by the host processor shall eliminate the spurious key if there is any. However, the chance is extremely small, the host processor is not wasting its time in dealing with any unnecessary data.

CHAPTER 5

A NEW HASH BASED JOIN ALGORITHM

In the previous chapter, the hashed bit array store filtering is mentioned since the architecture of the DBCP is designed to support the filtering technique. The first section of this chapter discusses more about the hashed bit array store technique and it's problems. The second section explains how the stack oriented filter technique (SOFT) improves the filtering effect. Based on the SOFT, a new join algorithm is developed and illustrated in the third section. The simulation of a new join algorithm is performed and the results and statistics are shown in the fourth section.

5.1 Limitation of Hashed Bit Array Store Technique

A problem associated with the hashing technique, called hashing collision, occurs when more than one key applied by the same hash function generates the same hash address. To minimize this problem, more than one functionally different hash coders are used as explained in the previous chapter. The major problem of HBAST is that when the source relation (smaller than the target relation) is large, one shot filtering scheme of the hashed bit array stores does not eliminate all of unwanted data so that the host processor has to carry the burden.

The following discussion shows how the problem of hashed bit array store filtering technique arise using the DBCP architecture. ~~use the soft~~. As shown in Figure 4-7, the DBCP has (each join attribute value) five different hash coders. Thus, each key value is transformed into five hash addresses by these five functionally different hash coders, and those five hash addresses are used to mark the corresponding bit array stores. Since the chance that two different key will have five pairs of identical hashed values using

five statistically independent mapping hash functions is extremely small. However, the hashed bit array store alone still allows for existence of spurious keys. This problem can be explained for better understanding with Figure 5-1 which shows that key K1 is mapped to five addresses denoted by M1(H1(K1)), M2(H2(K1)), M3(H3(K1)), M4(H4(K1)), and M5(H5(K1)). The next tuple in the source relation A contains key K2, which is mapped to M1(H1(K2)), M2(H2(K2)), M3(H3(K2)), M4(H4(K2)), and M5(H5(K2)). Key K3 in a tuple of the target relation B is transformed by the same hashing functions to generate five hash addresses. Then the bits in M1(H1(K3)), M2(H2(K3)), M3(H3(K3)), M4(H4(K3)), and M5(H5(K3)) are read. Unfortunately, all five bits are examined to be '1', due to the bits set by mappings of K1 and K2. Thus, the K3 is regarded as a join attribute which is matched with a join attribute of some particular tuple in the source relation A. However, K3 is neither K1 nor K3, but it is a spurious key.

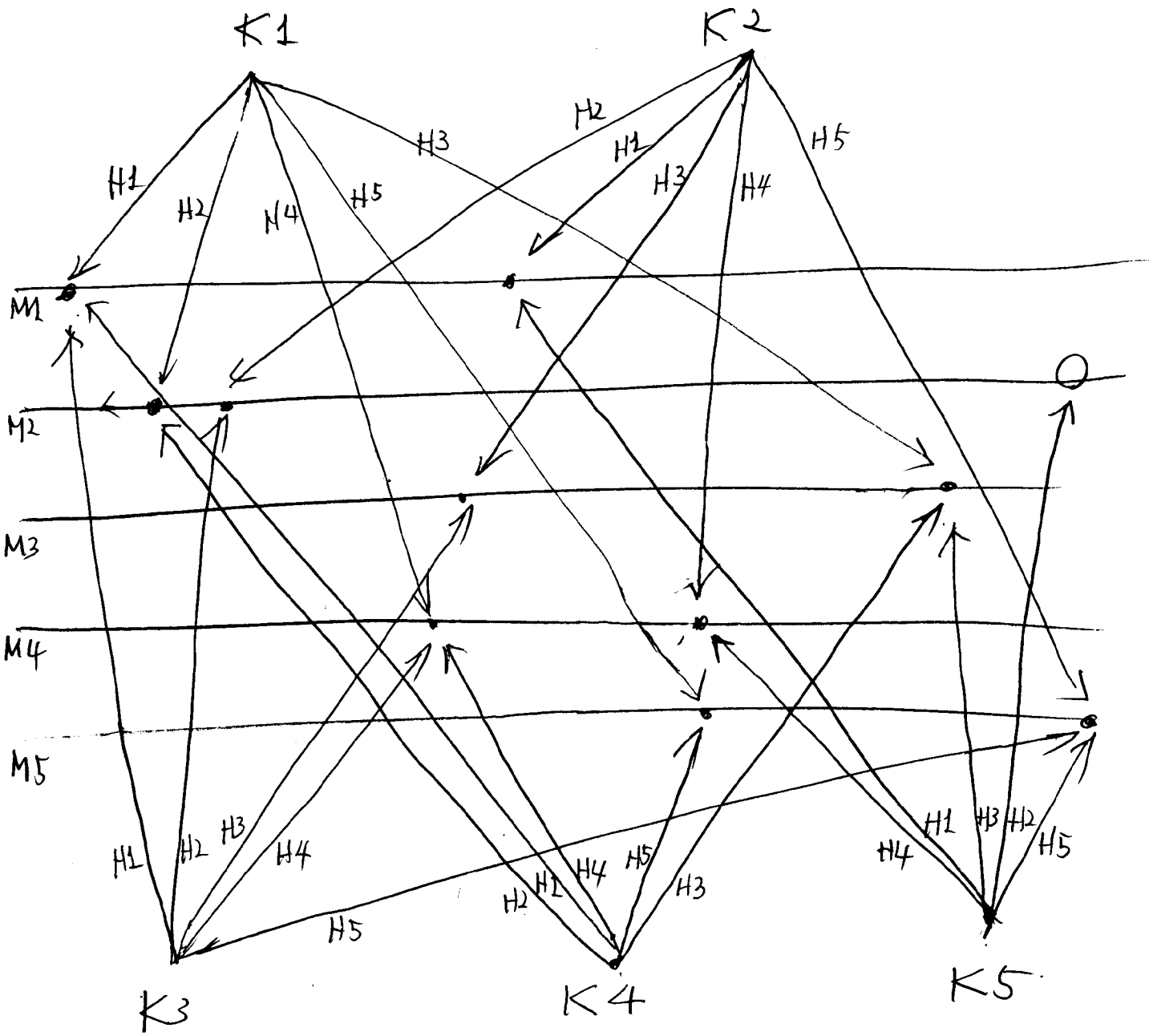


Figure 5-1 Key Mappings.

The key K4 is, on the contrary, is a wanted key, and it's associated tuple will highly probably be concatenated with the tuple of the key K1 by the host processor after a final screening. The key K5 will be discarded as soon as the logical AND value of five bits from the bit array stores is detected to be zero due to M2(H2(K5)) = '0'.

There will be more spurious keys when many bits in M1, M2, M3, M4, and M5 bit array stores become '1' after a large size source relation has been scanned. Therefore, this problem is data size dependent problem. If the number of tuples in the source relation is much greater than the number of bits in a bit array store, the number of filtered target tuples will be drastically reduced. Accordingly, the performance of the hashed bit array bit array filtering technique can be reduced for some large inputs. This problem is solved by the stack oriented filter technique (SOFIT) which is explained in detail in the next section.

<skip this>
data dependency

hashed, and the hash addressed bits in corresponding BASs are examined to determine whether the tuple is to be stored in the hash table or if is to be discarded. The remaining procedures are the same with the previous case of sufficient memory. After one subset file is finished, another subset file is processed until no subset file is left.

5.5.2 Union

The hash coder in the DBCP can be utilized efficiently in performing UNION operation.

$$R1 \cup R2 = R$$

$$\begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cup \begin{bmatrix} 6 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \end{bmatrix}$$

Figure 5-X. Example of UNION operation.

~~As shown in Figure 5-X~~, source relation (R1) is first read from the secondary storage; If it does not fit in main memory, it should be divided by the the first hash coder into subset files ($S_0, S_1, \cdots, S_N$). By the same hash coder, target relation (R2) is also divided into subset files ($T_0, T_1, \cdots, T_N$). After the division process, the tuples in the first source subset file (S_0) are hashed by the five hash coders in the DBCP, (Some portion of a tuple is used for key if a tuple is too long).

If there is sufficient main memory space, it is not necessary to divide the relations, and the same process for each subset file is applied to the whole input file which can fit in main memory.

their corresponding bits in the BASs are set, and the tuples are stored in hash-addressed bucket in MM. Then the tuples in the first target subset file (To) are hashed by the same five hash coders in the PBCP, and the five bits in the five BASs are examined to see the probability of duplicacy. If all of those five bits are set, it is probable to have an identical tuple in the To. So the hash-addressed bucket by the first hash coder is searched for a match. In this process, the whole tuple is compared as a key. If there is any match, the target tuple is (e.g. 5 and 1 b in Figure 5-X) eliminated immediately. Otherwise, the target tuple should be included in the resulting relation. After the last tuple in the target subset file is processed, the contents of the five BASs are cleared for the second subset files of source and target relations. The same process repeats for the second subset files and all others.

5.5.3 Difference

The difference relational database operation, executional process of is similar to that of the union operation:

When a duplicate (a match) is found, both source and target tuple is eliminated from the addressed-bucket. Then the tuples left in the hash table compose of a resulting relation.

$$\begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} - \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 3 & a \\ 9 & a \\ 2 & b \\ 4 & b \end{bmatrix}$$

5.5.4 Intersection

execution process of
The intersection operation is similar to that of the join operation. The partial tuple is considered as a key if the tuple is too long. If the source relation does not fit in main memory, it has to be divided using buffers based on the hash addresses produced by the first hash coder. Every tuples in a source subset file (S_0) are hashed and the hash-address bits in the five BASs are marked. Then the tuples in a target subset file (T_0) are hashed and the corresponding bits in the BASs are examined. If at least one bit is not set, the target tuple does not have any potential to be included in the resulting relation so that it is eliminated right away. Otherwise, the hash-addressed bucket is probed for a match. If there is a match (an identical tuple is found),

$$\begin{bmatrix} 5 & a \\ 3 & a \\ 9 & a \\ 1 & b \\ 2 & b \\ 4 & b \end{bmatrix} \cap \begin{bmatrix} 5 & a \\ 10 & b \\ 15 & c \\ 2 & d \\ 6 & a \\ 1 & b \end{bmatrix} = \begin{bmatrix} 5a \\ 1b \end{bmatrix}$$

Figure 5-X Example of Intersection operation.

the tuple is included in the resulting relation. (e.g. 5a and 1b in Figure 5-X) After the process for the first pair of subset files (S0 and T0), the five BASs are cleared and the source tuples in the hash table are all eliminated. The same procedure repeats for the second pair of subset files (S1 and T1) and all other pairs. The matched tuples are accumulated in the resulting relation during those processes.

CHAPTER 6

performance and comparison.

SUMMARY AND CONCLUSIONS

3.6 Summary

The previous sections address the necessity of database computer development since conventional computers are not inherently optimized for nonnumeric processing. The background research has shown three major database models and their physical data structures and operations. Due to the advantages of the relational data model, it has been chosen for most of the database machines. The major problem with relational database machines, however, is the frequently used and time-consuming join operation. Accelerating the join operation is the key to improving the performance of relational database systems.

As described in the background, there are three major join algorithms: the nested-loop algorithm, the sort-merge algorithm, and the hash-based algorithm. The nested-loop and sort-merge algorithms were used for many database computers during the early stages of database machine development. After Babb's hashed bit array stores <BABB1> became known to the public, people started recognizing how important the filtering technique is <DEWI3, QADA2, SHAP1, VALD2>. Combining hashing and filtering techniques seems to be an ideal approach. Goodman <GOOD1> and several other researchers have taken advantage of Babb's hashed bit array stores filtering technique, but the ideal approach has not been fully explored yet.

The dissertation, based on this ideal approach, ~~will~~ discusses the highly modular relational database computer equipped

but it is not excluded for the further research. (2)

with a single chip back end processor for the join operation. Parallel multiprocessing has not been chosen due to its complex synchronization problems and lack of cost effectiveness. A single join back end processor with specialized hardware that maximizes the filtering effect during the hashing process ~~looks promising~~. filter device has been developed. ← D

Since the proposed join back end processor basically uses a hash-based join algorithm, hash functions such as the simple division hash method, which is generally accepted as superior, need to be investigated before being adapted. It is assumed that any necessary hardware aids are supplied during the implementation of the hashing process to speed up the address calculation time. The results of the experiments performed on the hash algorithms will be included in the dissertation.

Based on the hash experimentation results, a hash method will be selected. Then, a join algorithm will be developed using the chosen hash method. The join operation in this database machine will be simulated to prove the logical correctness, and the necessary statistics will be provided at the same time.

The two different hardware and software back end approaches will be examined and compared to offer a general idea as to whether to build a new microprocessor or to implement the proposed join algorithm on the existing MC68030 microprocessor, used as a software back end processor. In both cases, the execution times for the chosen hash algorithm will be provided in order to make a better design decision.

① The join back end processor is a stack oriented filter device. Within it, five statistically independent hash coders have associated bit array store. ~~The bit array store~~ (BAS) is used either ~~for indices to buckets or for hashed bit array store filtering technique~~. The stack pointer always points to the current BAS. The BASs which are below the current BAS are saved in the stack for later use as records for non-empty buckets. The current BAS and the BASs above the current one participate in hashed bit array store filtering process. The hash coder which is attached to the current BAS produces hash addresses for the tuples to be stored in the addressed bucket. Thus, the five BASs are used as elements of a stack so that the stack oriented filter technique (SOFT) become feasible. The new hash-based join algorithm is developed based on this technique. The distinguishable difference between the new join algorithm and

Other hash-based join algorithms such as straight forward, simple, GRACE, and hybrid is that in every division process, of the new join unnecessary data are detected and filtered by the hashed bit array filtering technique while other join algorithms carry around unwanted tuples all the way until the last moment of join attribute comparisons. The new hash-based join algorithm repeats this division and filtering process many times in recursive way, so almost 100 percent necessary data are left (or sent to host processor) for final screening and merge. ~~In this method, the number of comparisons are drastically reduced,~~ Those transmitted source and target tuples at a time are almost always merged together to produce output for resulting relation.

Since after repetitive division and filtering processes, the remaining tuples in source and corresponding

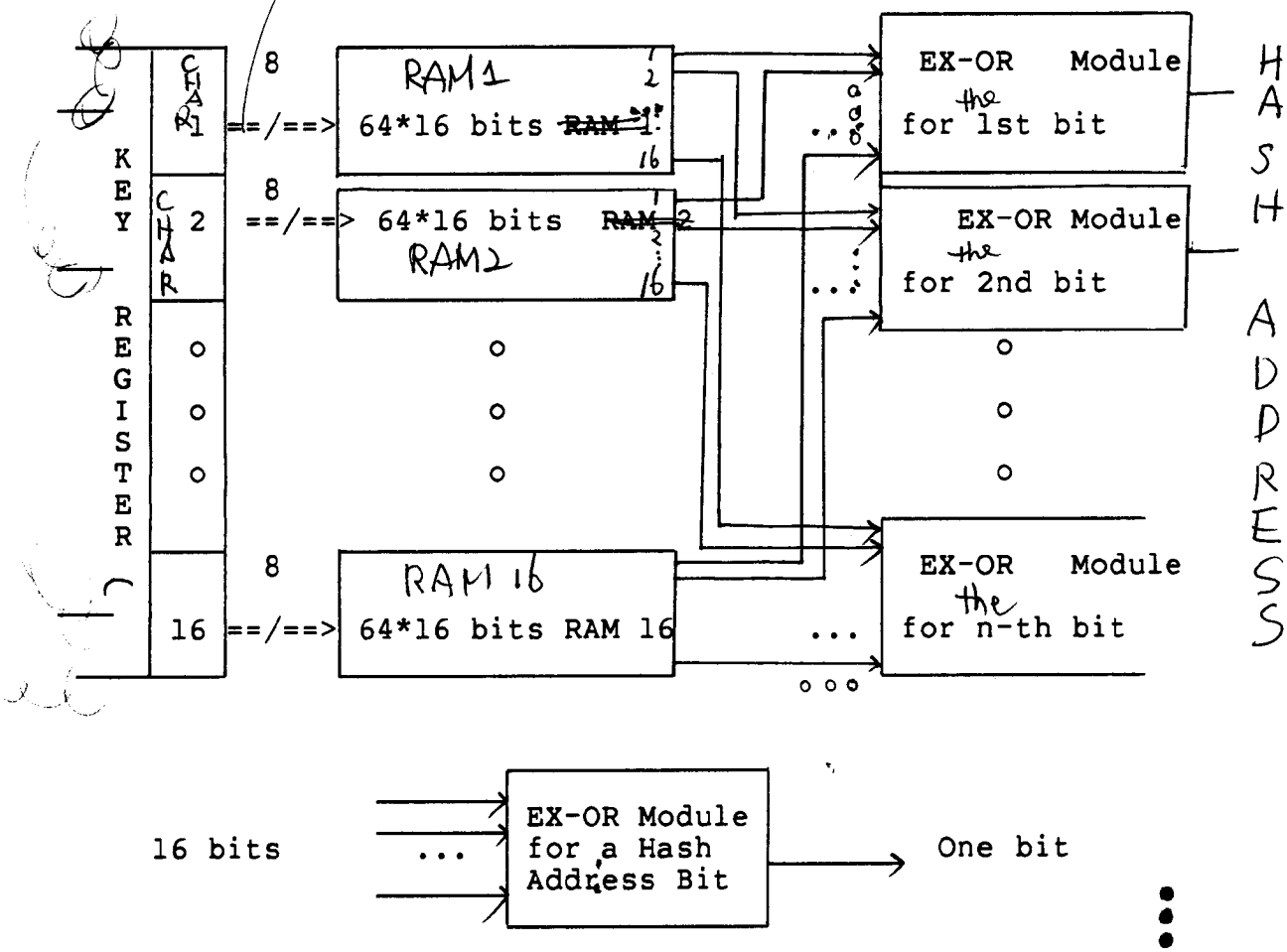


Figure 3-2 The Hardware Mapping Hash Coder

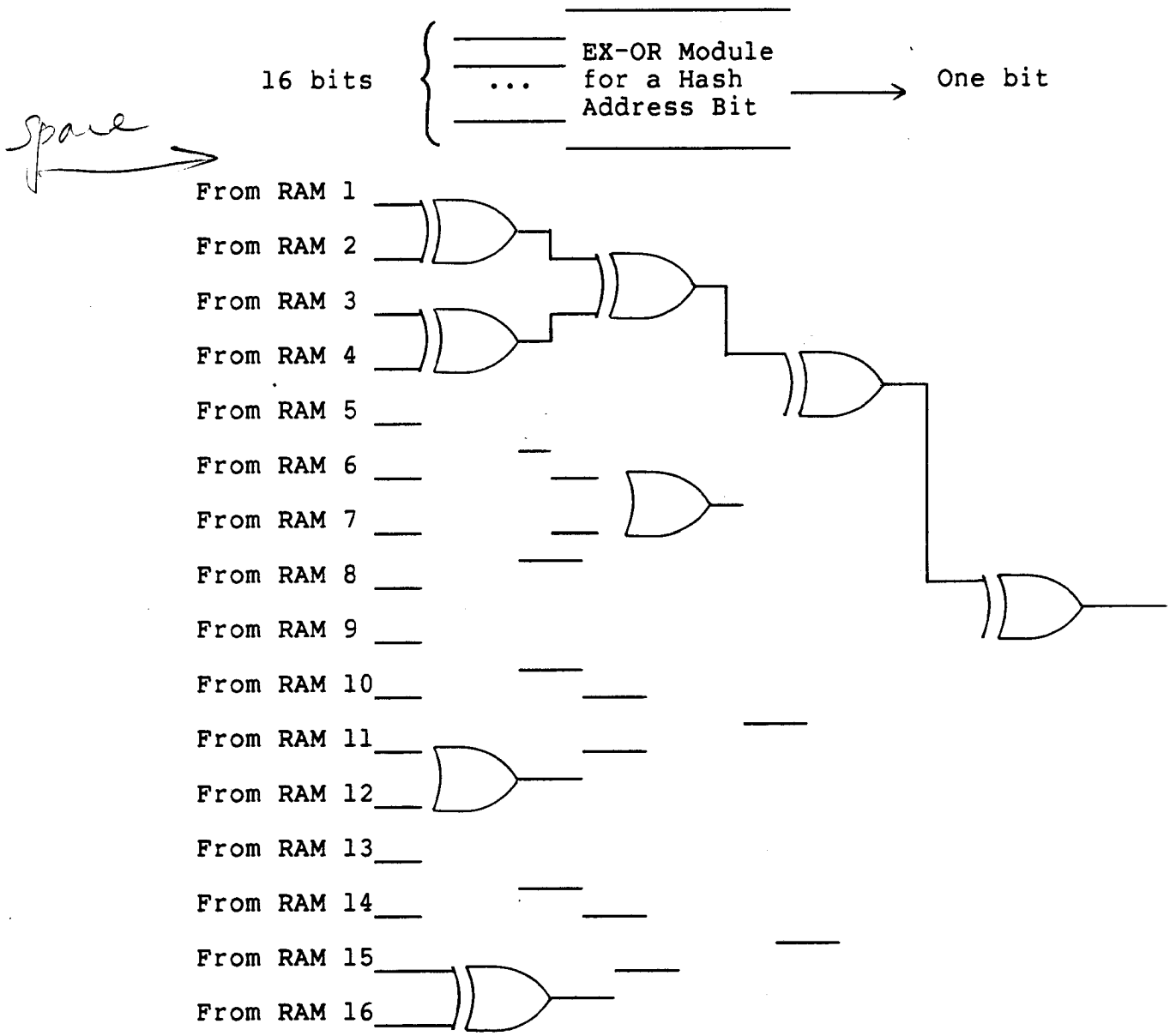


Figure 3-3 EX-OR Module for a Hash Address Bit

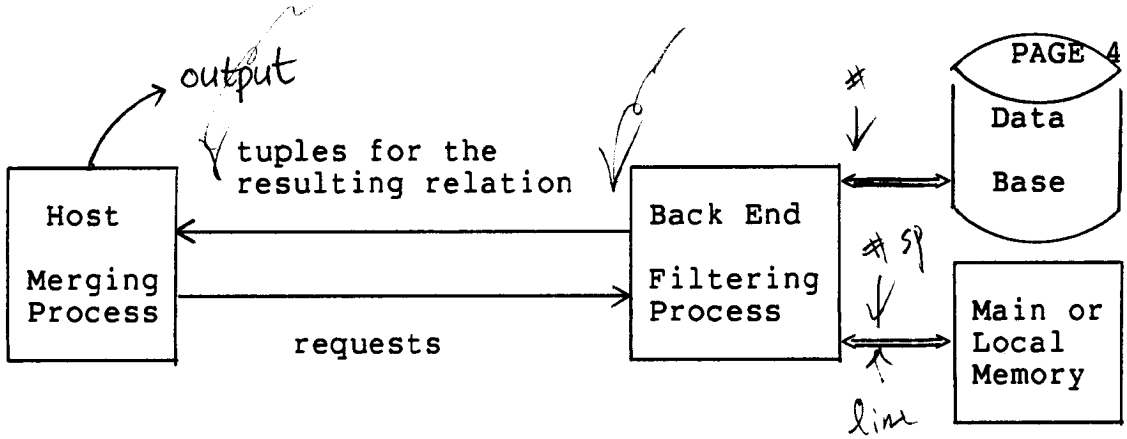


Figure 4-1 Example of the Execution of Relational Join

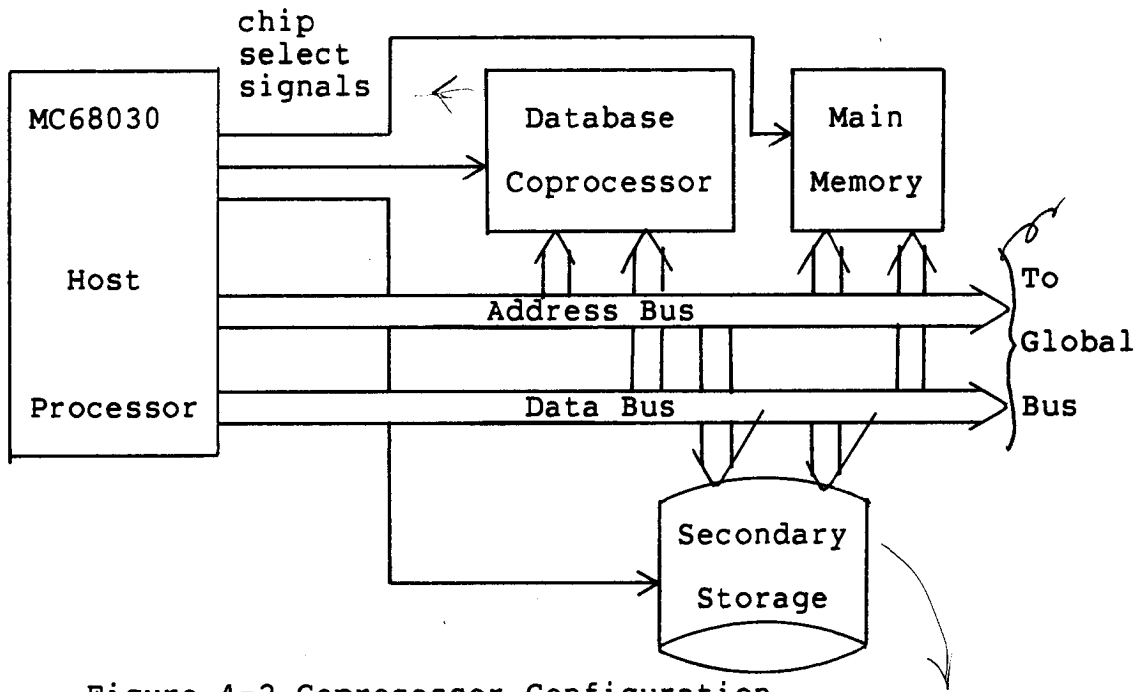


Figure 4-2 Coprocessor Configuration

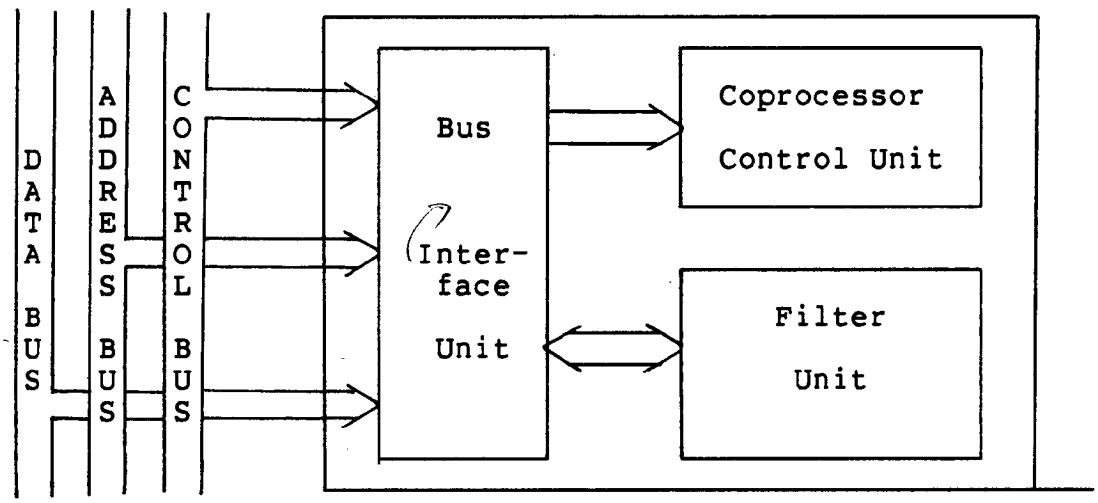


Figure 4-5 DBCP Simplified Block Diagram

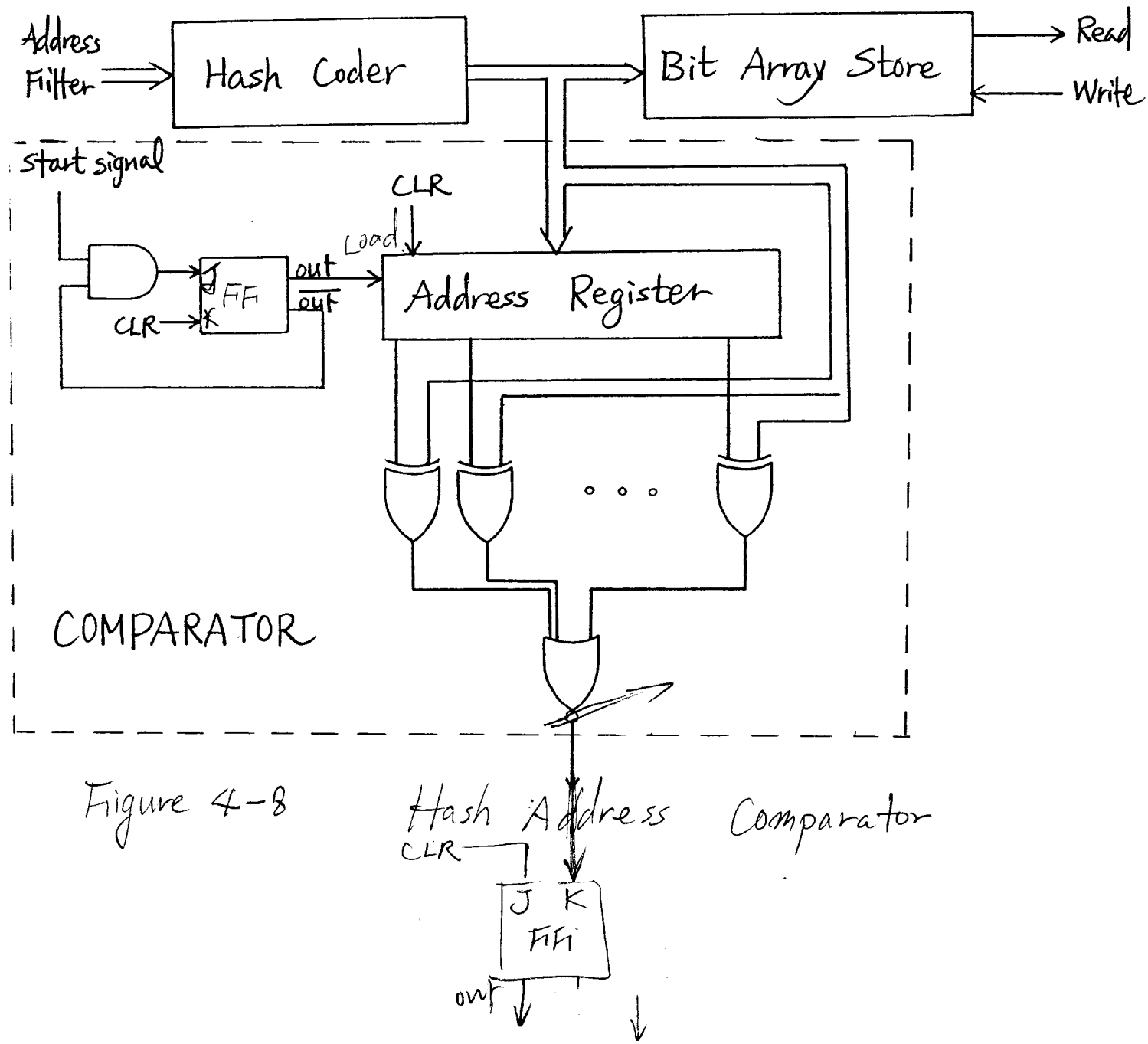

Address Filter
Hash Coder
Bit Array Store
Read
Write
start signal
CLR
Load
FF
out
out
CLR
Address Register
COMPARATOR
Hash Address Comparator
CLR
J K
FF
out

Figure 4-8

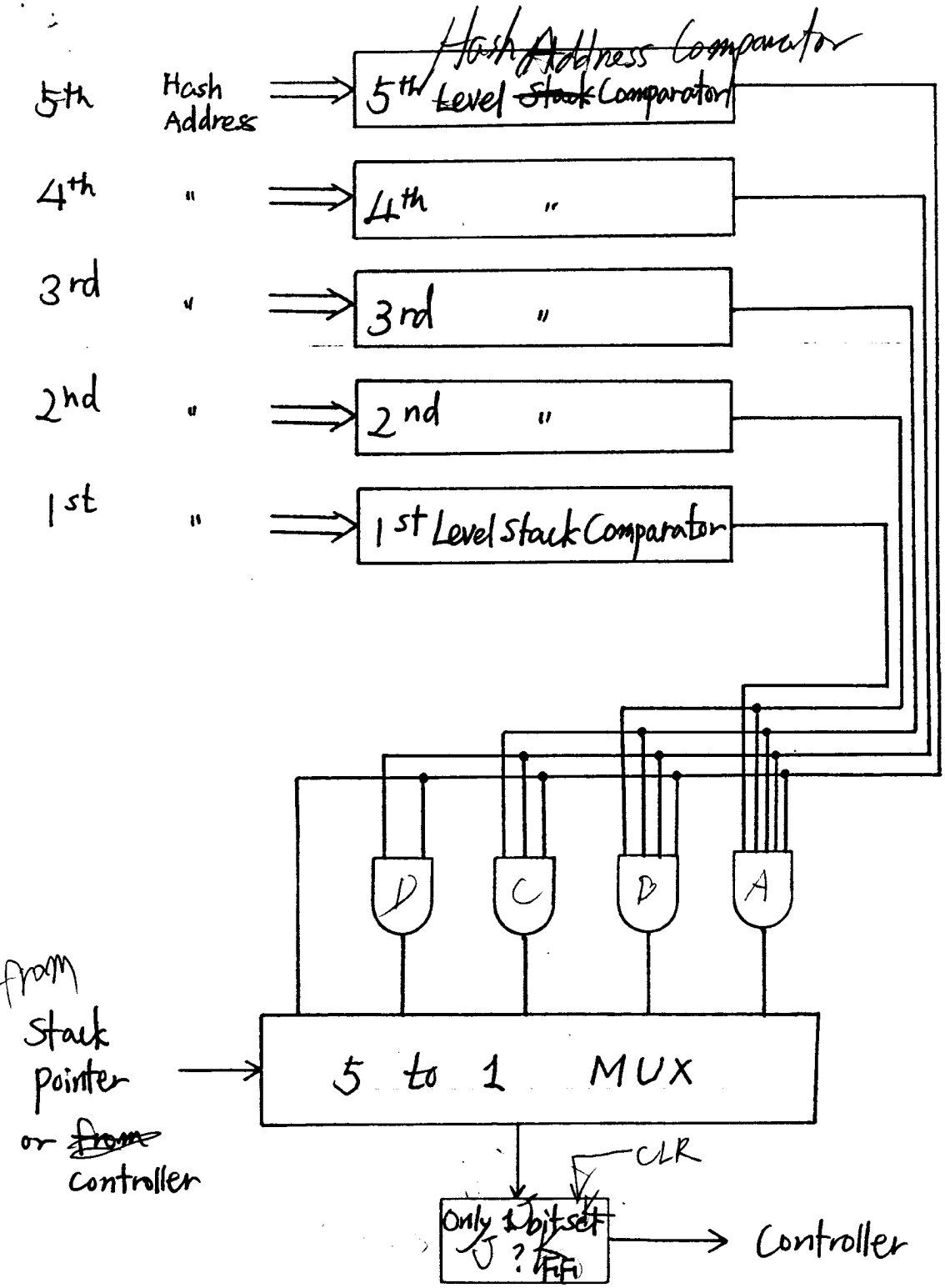
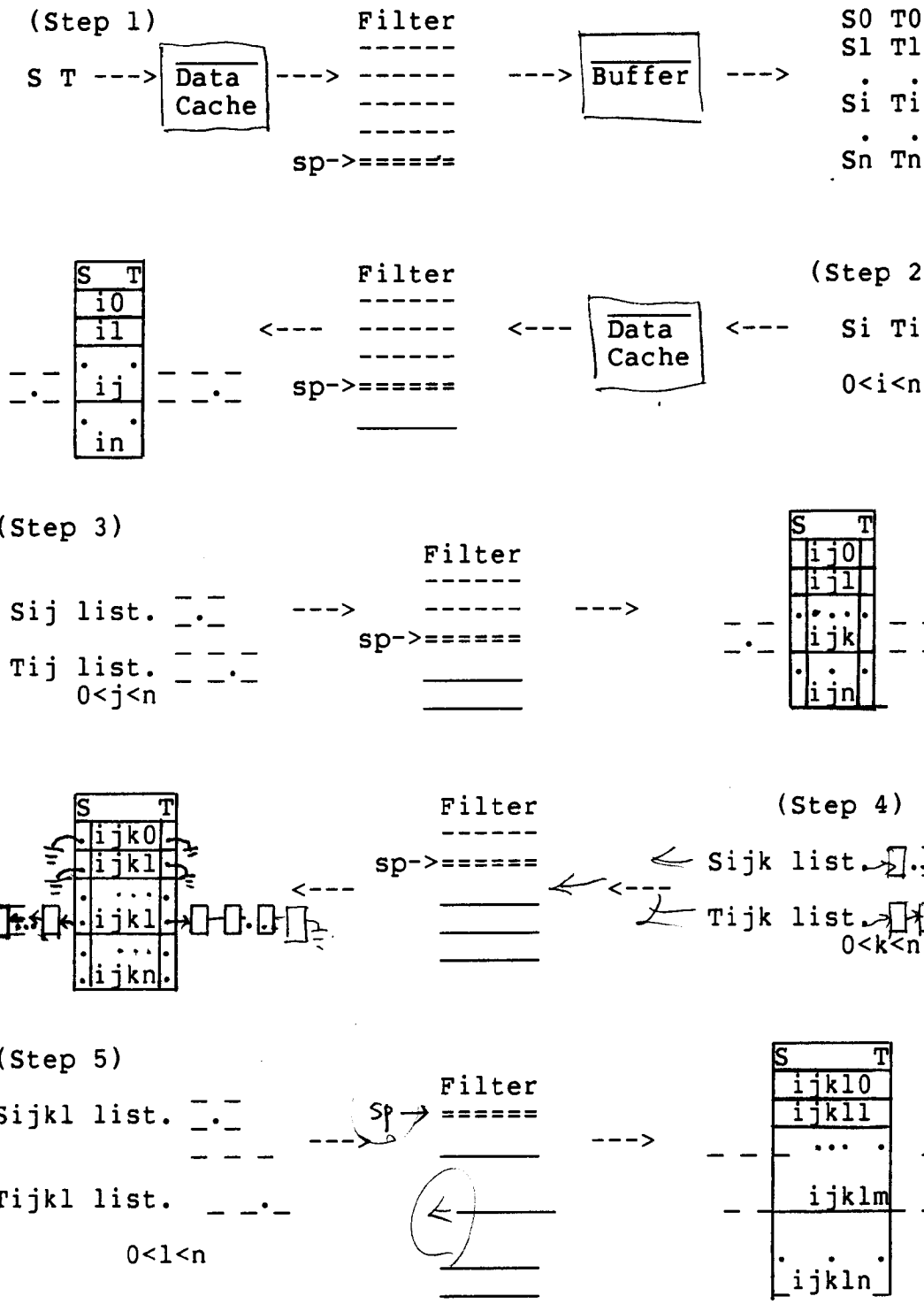


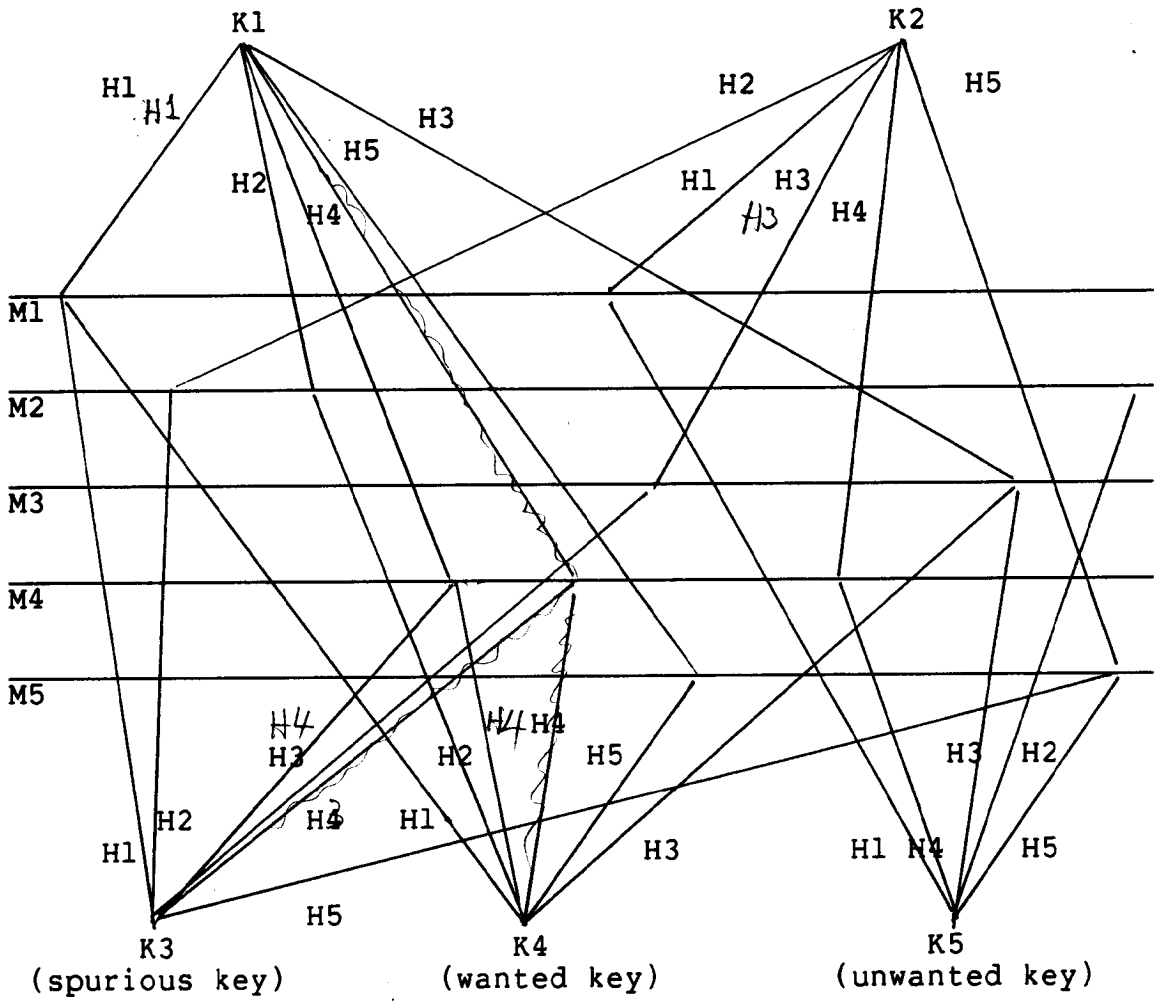
Figure 4-9 Hardware for Stack Oriented Filter Technique (SOFT)


(Step 1)
S T
Data Cache
Filter
sp->
Buffer
S0 T0
S1 T1
Si Ti
Sn Tn
S T
i0
i1
ij
in
(Step 2)
Si Ti
0<i<n
(Step 3)
Sij list.
Tij list.
0<j<n
ij0
ij1
ijk
ijn
(Step 4)
Sijk list.
Tijk list.
0<k<n
ijk0
ijk1
ijkl
ijkn
(Step 5)
Sijkl list.
Tijkl list.
0<l<n
ijkl0
ijkl1
ijklm
ijkln
(Step 6)
.Sijklm list
.Tijklm list
To the host processor <---

$0<m<n$

Figure 5-3 Join Process in the SOFT

Keys of Source Relation A



Keys of target relation B

Figure 7. Key Mappings

target list has extremely high probability of having identical join attributes and all others are eliminated before any unnecessary comparison of join attributes, so the number of join attribute comparisons is drastically reduced. As a result, total data movements in performing a join are radically diminished.

The output of simulation program has shown that the new hash-based join algorithm is logically correct. And the number of tuples passed through the processor in the simulation manifests how effective data movements of the new join method have been cut down comparing with those of conventional way of join.

The new join method can be divided into two processes such as filtering process and merging process with final screening. In HIMOD database computer, the filtering process is performed by the database coprocessor (DBCP),

① use of CAFIS filter device will probably improve the overall system throughput.

Although CAFIS's hashed bit array store filtering technique is adapted by the HIMOD →

Since CAFIS has wider range of bit array store (64K bits), many unnecessary data might be eliminated while they are moved from a file in secondary storage to main memory.

As several researchers experienced CAFIS's demonstration of dramatic improvement in all join algorithms <DEWI3, GADA2, SHAP1, VALD2, SCHN1>, with an aid of CAFIS, the HIMOD may provide even faster join.

and the merging process is executed by the host processor whenever it receives source and target lists of tuples from the DBCP. The architecture of the hardware back end (DBCP) has been illustrated in chapter 4 to show how the stack oriented filter device is designed to filter unwanted data efficiently. → next page

The HZMOD uses a Motorolla 68030 (MC68030) microprocessor as the host processor, and the DBCP communicates with the host processor through a protocol, defined as the M68000 coprocessor interface <MOTO1>. The DBCP can be designed as either a software back end or a hardware back end. For a software back end, a MC68030 can be used and the filtering process of the new join algorithm must be implemented in software later, but filtering process for each join attribute takes about 40 times longer based on the calculation of hashing speed. than hardware back end

(1) <BUCK1> <KUM1>

Buckholz and Lum reviewed hash functions, and they recommended division method as the best one. Knuth later concluded that ~~even though many methods have been suggested,~~ none of ~~them~~ hash methods has proved to be superior to the simple division and multiplication methods. People generally accept this statement as true when distribution performance is essential while hash address calculation time is not.

It is heavily emphasized that the algorithm compute hash addresses very fast using any necessary hardware components since a huge amount of data has to pass through the hash coder in the DBCP. To speed up the hash address computation, main efforts in designing a new hash function are avoiding time-consuming serial and/or iterative computations while taking

The major operation of the DBCP in filtering data is hashing, and five hash coders parallely produce five hash addresses. Each hash function should be statistically independent so that the five hash addresses from a key must not be related each other. ~~To provide such a hash function for the hash coders~~, Most of the well known hash functions, including mapping, shift-fold-loading, Hu-Tucker Code, Fold-by and several new hash functions are surveyed in this dissertation. It is assumed that any necessary hardware aids are supplied during the implementation of the hashing process to speed up the address calculation time. ~~Each~~ hash function has been simulated and applied to two name data sets and one numeric string data sets to produce distribution performances in terms of mean square deviations. The speed of calculating a hash address is measured for each hash in terms of clock cycles

advantage of parallel processing by means of hardware in generating each hash address. And, of course, the new join algorithm should distribute keys into buckets as uniformly as possible. ~~and~~ types of

Therefore, the ideal hash function that I have attempted to design is a data-independent hash function that calculates a hash address within few machine cycles with relatively good distribution. The new mapping hash function is the one that satisfies those requirements if it is implemented in hardware.

function in both hardware and software implementation cases. Then the cost of hardware implemented hash coder is calculated and provided in terms of number of gates.

As the results have been shown in the Table 3-1, the mapping hash method satisfies all three requirements — of the very highest rank. Moreover, it is advantageous in generating statically independent hash addresses in the same period of address calculation time by keeping different sets of prime numbers for each hash coder. The fold-shifting hash method like FS(0, 10, 20, 30) and FS(0, 11, 22, 25) has a problem in finding three more similar type of statistically indepent hash functions with relatively good distribution, but it is very fast and inexpensive.

Consequently this dissertation research provides a new join algorithm and new hash functions. The new join algorithm will shorten the time needed for a join since it goes through frequent filtering processes to discard unnessary data. The more main memory space is allowed, the more this join method will be effective. The new mapping hash function in hardware hash cele implemented will help speeding up the hash-based join algorithm since parallel processing is used in the hardware hashcoder to calculate a hash address in 3 clock cycles. And the mapping hash method distributes keys relatively good comparing with other well known methods.

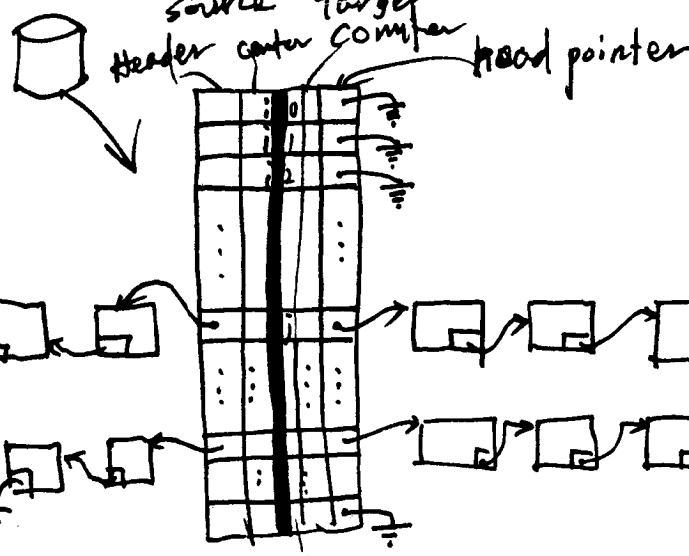
Discriminator	No of Tuples in Source	Target	Result	No of tuples brought into the processor. Join using the SORT	Conventional Join (Nested-loop)
A	155	1893	355	2426	293415
E	646	1402	919	3795	905692
G	799	1249	902	4095	997951
K	1196	852	846	4419	1018992
V	1961	87	205	2902	170607
Table 5-1 SUM				17637	3386657

The number of tuples brought into the processor is selected as the major measurement of the overall performance although there are other factors to be considered. Since more data movements creates system overheads and slows the join operation, the less number of tuples are brought into the CPU, the shorter response time might be taken for the join.

The new hash-based join using the SOFIT takes about two hundred times less data movements than the conventional nested-loop join method in average. (3386651/17637) The biggest reason for this constasting performances is that the SOFIT eliminates all of the unnecessary data (about 99.999999999999 % of those) in filtering process while dividing the source and targets relations into groups of tuples such that source tuples in a group can be matched only with some target tuples in a corresponding group. Removing unnecessary data while hashing and diriding certainly helps reducing data movements drastically. On the contrary, the conventional nested-loop and sort-merge carry around all the data including unwanted data, and discard in the very last moment that they find it is unnecessary data by direct comparison of join attributes.

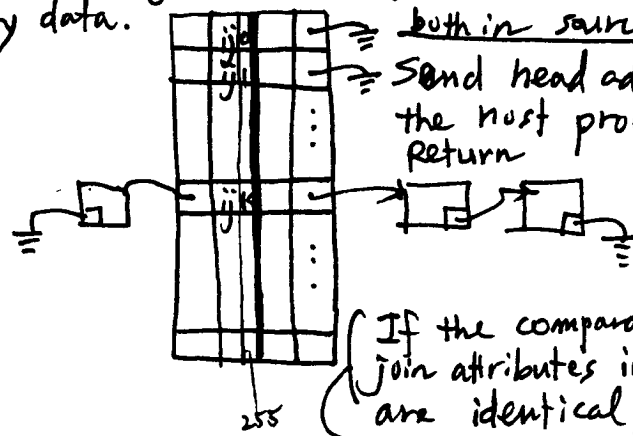
The time complexity of the new join algorithm is $O(S+T+R)$ (or $O(N)$), but this asymptotic notation might not be the only one heavily reflected in actual response time due to other influencing factors. However, the new hash-based join algorithm might outperform other hash-based join algorithms such as simple, GRACE, and Hybrid because none of them include the concept of filtering in their algorithms. Their common method to find a matched source tuple for an incoming target tuple is to compare the join attribute of the target tuple with every source join attribute in the hash-addressed bucket one by one. This repeated direct comparisons might slow the system, but in the new join method, the direct comparison of the join attributes is only allowed after all of the unwanted data are casted out for the final screening for merge.

Hash i-th subset files



Step 1

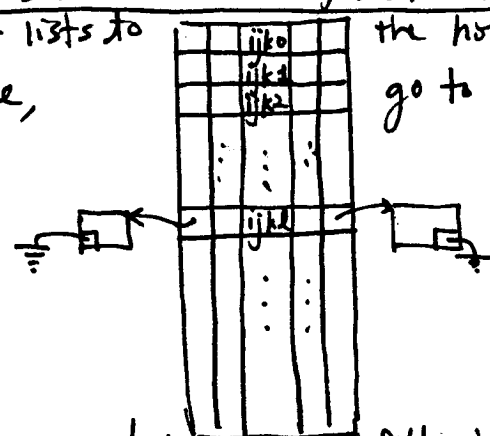
Hash ij-th source & target lists, filtering unnecessary data. If there exists only one list both in source & target directory, send head address of the lists to the host processor for a merge, and Return



Step 2

If the comparator determines that all the join attributes in the source & target lists are identical,

Hash ijk-th source & target lists filtering unnecessary data. If there exists only one list both in source and in target directory, send the head addresses of the source & target lists to the host processor for a merge, and Return. Otherwise, go to step 4 to step 2.



Step 3

Hash ijkm-th source and target lists filtering unnecessary data. If the comparator determines that all the join attributes in the source and target lists are identical, send the head addresses of the source and target lists to the host processor for a merge and return to the previous step. Otherwise, go to the next step for further division process.

5.4 Simulation Results: A Comparison with the Conventional Join

The simulation was performed on an IBM 4381 mainframe computer, and the listing of simulation program in Pascal is in Appendix I. The set resulting from the combination of the 1024 generally chosen names (GCN) data set and the 1024 randomly chosen names (RCN) data set is used as one data set. Both name data sets are the same data sets used in the experimentation for hash functions. The combined data set contain 2048 name tuples, which are read into the system to create the source relation and target relation. While each tuple is scanned, the initial letter of the last name is compared with 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 are 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, while the whole name is used to produce a hash address in the