Order Number 9128428

# A comparative study of hash functions for a new hash-based relational join algorithm

Shin, Dong Keun, D.Sc.

The George Washington University, 1991



U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

# 2. Final Dissertation

# A COMPARATIVE STUDY OF HASH FUNCTIONS FOR A NEW HASH-BASED RELATIONAL JOIN ALGORITHM

by

Dong Keun Shin

B.A. Computer Science,
The University of California, Berkeley

M.S. Computer Science,
The George Washington University

A Dissertation submitted to

The Faculty of

Department of Electrical Engineering and Computer Science
The School of Engineering and Applied Science
The George Washington University

in partial satisfaction of the requirements for
the degree of Doctor of Science in Computer Science

Supervised by

Dr. Arnold C. Meltzer

Professor of Engineering and Applied Science

A COMPARATIVE STUDY OF HASH FUNCTIONS FOR
A NEW HASH-BASED RELATIONAL JOIN ALGORITHM


ABSTRACT

Among the several relational database operations, the join operation is the most time-consuming operation; nonetheless, it is the most frequently used operation. Although many researchers have endeavored to increase the speed of the join, this issue remains less than fully explored. In this dissertation, the author's new hash-based join algorithm is developed and implemented on a hypothetical database machine, the 'Highly Modular Relational Database Computer (HIMOD).' The database computer is equipped with a general purpose processor as a host and a database coprocessor (DBCP) as a back-end processor. Before comparing the join attributes for merge, the new join method attempts to filter nearly all the unnecessary data as early as possible. Simulation results show that the new hash-based join method takes about two hundred times less data movements than the conventional nested-loop join method. Therefore, the DBCP is designed to be a rapid filter device.

The major operation of the new hash-based join in the DBCP is hashing; therefore, the study developed a new, fast, hardware oriented hash function to serve as a hash coder of the DBCP and speed up filtering. The new hash algorithm and

the new hash-based join are designed to take advantage of parallel processing. When both algorithms are implemented on the parallel architecture hardware DBCP, the response time for the join operation can be drastically reduced.

This study surveyed several newly developed hash functions along with well-known hash functions such as algebraic coding, digit analysis, division, folding, midsquare, multiplicative, radix, random, and Pearson's table indexing. The comparative analysis of the hash coders was based on criteria such as speed, distribution, and cost. The author's new mapping hash method not only has reliable and relatively good key distribution, but it also takes only three clock cycles to calculate a hash address if the mapping hash coder is implemented in hardware. Therefore, the dissertation concludes that the new mapping hash method is the best one for the hardware hash coder of the DBCP and for similar applications.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

Chapter 1 will introduce the main objectives and scope of the dissertation. To that end, database machine architectures will be explained, and approaches taken by existing database computers to accelerate the join relational database operation will be described. This chapter also covers the issues in designing an effective hash coder for the relational database back-end since the hashing technique has become popular in performing join and other relational database operations.

## 1.1 Scope of Dissertation

Computers were first conceived to perform special computational tasks in research environments, but their use has grown rapidly and has become more widespread. In this information-oriented age, computers are much more involved in nonnumeric applications rather than numeric computations. The effective and efficient management of large quantities of data is of considerable importance to most large organizations, such as industries, scientific research centers, and military organizations. The growing need for data management has been accompanied by growing demand for high per-

formance computers to perform database management efficiently.

In recent years, integrated circuit technology has made surprising progress in lowering the prices of memory, processor, and I/O devices. As a result, computer hardware has become less expensive, making development of a special-purpose computer that is able to effectively manage database systems economically feasible. Such a special-purpose computer is generally referred to as a database computer or database machine. These computers are dedicated to performing some or all database management functions. The definition of a database computer includes distributed database management systems which are built as a layer of database management software with an interface to conventional computers. However, for the purposes of this dissertation, the term "database computer" refers to devices or processors specially designed to improve the performance of database management systems.

Most database computers have been constructed for relational databases. The advantages of the relational model over the hierarchical and network models have become increasingly well recognized and are illustrated in section 2.1.6. Relational database operations are also relatively easy to implement in hardware. Among relational operations, the join operation is the most time-consuming operation; nevertheless, 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. The problem of complexity and inefficiency of the join operation is the major bottleneck for relational database management systems. Researchers have examined and have developed special devices or processors to accelerate the join operation, but this research area has not been fully explored. The aforementioned problems and the technical and economical feasibility provide researchers the motivation needed to examine join algorithms that can potentially produce a faster join.

The purpose of this dissertation is to provide an efficient and effective method for accelerating the time-consuming join relational database operation. The parallel computer architecture that facilitates a faster join and a new join algorithm, which can perform best when using the architecture, are presented in this dissertation.

The existing join algorithms are illustrated in Chapter 2 in order to explain both the history of the join operation and the alternative ways of implementing the join operation. Hash-based join methods are discussed in detail in section 2.5, since the proposed join algorithm also is a hash-based join method.

Designing an effective hash coder is in high demand owing to the fact that hash functions are used in many other

database operations, as well as in other applications. The major operation of the new hash-based join is hashing, so a new fast and effective hash coder is essential in order to accelerate the join. In Chapter 3, several proposed new hashing functions, used for a join operation, are introduced and compared with current hash functions in terms of distribution, speed, and cost. Even though every application environment that uses a hash coder is different, the basic approaches and principles used in implementing a hash coder, as described in this dissertation, may provide direction for research or an option for people who are seeking a good hash coder.

The architecture of the join hardware back-end processor, using the MC68030 Enhanced 32-bit Microprocessor as the front-end processor, is presented in Chapter 4. The join software back-end, which also uses the MC68030 as the front-end processor, is explained and compared with the hardware back-end in terms of speed.

Having thoroughly researched the hash functions, I have concluded that none of the current hash functions meet the requirements of both extremely fast hash address calculation and relatively good key distribution. Thus the mapping hash method is selected for designing the hash coder in the join coprocessor. Based on the chosen hash method, in Chapter 5, a new hash-based join algorithm is illustrated, along with the simulation results, to show its advantages over the con-

ventional join and other hash-based join methods. Other relational operations that utilize the implemented hash coder such as project, union, intersect, and difference are also discussed in Chapter 5. Finally, a summary of the whole dissertation and conclusions regarding how to find a good hardware oriented hash function and an efficient join algorithm are provided in Chapter 6.

## 1.2 Database Machine Architectures

### 1.2.1 Definition of a Database Machine

The conventional approach to database management requires that the database management system, other application programs, and operating system all run concurrently 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 (or back-end). 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 processor is categorized as either a hardware or a software back-end. The processor is categorized as a hardware back-end if there is a hardware enhancement or

modification on the back-end processor. If the database machine depends only on the innovative software architecture on the back-end processor, which physically duplicates the same architecture as the front-end processor, it is then referred to as a software back-end.

The database computer discussed in this dissertation named the "Highly Modular Relational Database Computer (HIMOD)," uses a single back-end processor 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, it is referred to as a multiple back-end approach. Accordingly, database machines can be divided into the following categories: single software back-end, single hardware back-end, multiple software back-end, and multiple hardware back-end, all of which are based on their architectural configurations.

The multiple back-end approach is utilized by many database machines in order to increase system performance through 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 that the data are distributed to the processors in a parallel manner. This approach is often called a multiprocessor system with replication of functions; it is used 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 either one or a small number of functions efficiently. These processors can be either general-purpose processors (or software back-ends) or special-purpose processors (or hardware back-ends). 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 RDBM <AUER1>, SABRE <VALD1, VALD2>, and DBC <HSIA1>, including HIMOD, use this direction. The database back-end processor in HIMOD is especially dedicated to the join database operation that is often described as the most time-consuming, yet frequently used, operation. Because the join operation is one of the major bottlenecks for the relational database management system, the database coprocessor in HIMOD is designed to release this bottleneck which consequently accelerates the join. In Chapter 4, the architectures of both hardware and software back-ends are discussed and compared in further detail in order to determine the most effective processing for the join operation.

### 1.2.3 General Architecture of the Back-End Database Management Systems

The simplest form of the back-end database management system is shown in Figure 1-1 which indicates the connections between the host, the back-end, and the data base in the secondary storage. The relationship between the host and the back-end is often considered a master-slave coupling, in which 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, in most cases, it also controls access to the database.

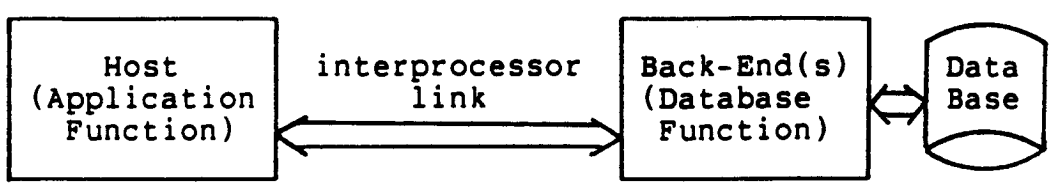


Figure 1-1. Back-End Database Management System

The interface sequence of application programs is as follows: the application program issues a service request to the back-end database management system that indicates the

operation required and the data to be operated upon. The back-end database management system also can request and receive service from the host. If the request from the host is validated, it is then served by the dedicated back-end processor.

### 1.2.4 Database Filter and Intelligent Secondary Storage Device Approaches

Besides classifying database machines according to their back-ends, database machines can be further classified according to the kind of problems (or limitations) the group of database computers attempt to solve (or remove). Some database computers exploit parallel processing capabilities that are built into the read/write mechanisms of secondary storage devices so that the data stored on these devices can be directly searched and manipulated. This technique allows the database computer to examine data locally before transferring the data to the central processing unit (CPU). As a result, this approach eliminates the limitations of the conventional storage devices in which the content of an entire relation or file must be brought from the secondary storage to the main memory for examination by the CPU; with a database computer, on the other hand, only relevant data is transferred to the CPU for further processing. Database computers, such as CASSM, RAP, and RARES <SMIT1>, accomplish

this local review by using 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.

In addition, slow mechanical movement in secondary storage devices limits the data transfer between these devices and main memory to the same speed as the storage devices. The database filter, which resides on the I/O channel, filters unwanted data from a requested file in secondary storage and sends only relevant data to the CPU. This method lessens traffic congestion in the I/O channel. This database filter approach is used by database machines such as CAFS <BABB1>, SURE <SU1>, and VERSO <HSIA1>.

## 1.3 Approaches to Increase the Speed of the Join Relational Database Operation

Ever since the relational data model was introduced by E. F. Codd's pioneering paper <CODD1> in 1970, the advantages of the relational data model over the hierarchical and network models have been increasingly well recognized. As stated previously, the relational join operation is both frequently used and time-consuming. The join operation merges 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 join condition.

In 1977, Blasgen and Eswaran <BLAS1> described several

methods for evaluating a general query, involving project, select, and join relational database operations. They compared these evaluative methods based on which method had fewer accesses to secondary storage. In their examination of the join operation, both nested-loop and sort-merge algorithms were analyzed and discussed. Because of the work of these two authors, researchers were generally convinced that a nested-loop join algorithm performed acceptably on small or large sized relations when a suitable index existed. Moreover, Blasgen and Eswaran concluded that a sort-merge join algorithm would be the choice when no suitable index existed. Both nested-loop and sort-merge join algorithms and their actual implementations are described and discussed in greater detail in Chapter 2.

Based on both nested-loop and sort-merge join methods, unnecessary tuples that are not necessary to the resulting relation for the join are still included until the last moment although they are not necessary to the resulting relation. Assuming that the amount of data in the source and target relations is large, but the amount of resulting tuples are relatively small, then most of the tuples in the source and target relations are not needed in producing the output for the join. However, all of those irrelevant tuples are also brought to main memory from secondary storage via the I/O channel; as a consequence the channel becomes congested, which, in turn, creates the aforementioned I/O bot-

tleneck. Many researchers have broached designing a database filter for the join operation to reduce the problem of channel congestion. Several database computers such as CAFS <BABB1>, SURE <SU1>, VERSO <HSIA1>, and DBC <HSIA1> have been designed based on the concept of database filtering.

The hashed address bit array stores filtering technique used in the CAFS (Content Addressable File Store) filter device demonstrated dramatic improvement in all join algorithms without substantially increasing hardware cost <DEWI3, QADA2, SHAP1, VALD2, SCHN1>. The hashed address bit array stores technique in CAFS uses single-bit array stores which are single-bit wide random access memory. The CAFS 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 in parallel. Then the CAFS 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, CAFS 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, in this way the potential matched tuples are sent to the host computer to produce the tuples of the resulting relation. Meanwhile, the unwanted

tuples are discarded, since they will not be included in the resulting relation. Among the potential matched tuples sent to the host processor, there may be spurious keys; therefore, the final screening is left to the host processor, which must compare the two join attribute values and then 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. Thus if the smaller source relation is so large that most of the bits in the bit array stores are set, the number of filtered target tuples might be reduced. Therefore, the performance of this technique is, to a varying degree, dependent on the size of the 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 further explained in Chapter 5, where the new join algorithm is introduced.

Since the cost of the main memory has been substantially reduced, hash join algorithms have been recognized as having a great potential. It is now a well-known fact that the join algorithm based on hashing is more advantageous than nested-loop or sort-merge join algorithms: a fact that has been noted by <DEWI3>. One noticeable difference of hash-based join algorithms is that they minimize the amount of data moved during the process of executing a join algor-

ithm. Join algorithms, such as nested-loop and sort-merge, require frequent key comparisons which results in more data movements. The three major hash join algorithms are illustrated in section 2.5, while time complexities are discussed in section 2.6.

## 1.4 Issues in Designing an Effective Hash Coder for a Database Back-End

The new join database coprocessor (DBCP) uses a hash-based join algorithm to perform a faster join; therefore, the major operation in the DBCP is hashing. Since millions of bytes might be hashed out through a hash coder in the DBCP, designing an effective hash coder in the DBCP has been another important goal of this research. Hashing has been a fruitful research area since the early sixties, as noted by Buchholz <BUCH1>. Major articles published between 1971 and 1975 <MAUR1, LUM1, KNOT1> have introduced hash functions and have analyzed their performances.

However, although these hash functions have been carefully examined, none of them fully meets the requirements for the application of the database filter. Assuming many tuples are needed to be hashed in a database computer, any hardware aid can be adapted for a hash coder to calculate hash addresses very fast. One of the concepts for fast calculation is the 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, negation, and shift operation are recommended procedures. Any time-consuming serial and/or iterative computations should be avoided as much as possible in order to reduce address calculation time.

It is worth noting that in the hardware back-end approach, the hash coder does not need to be similar to the arithmetic logic unit (ALU) of the host processor. The cost of adapting hardware components need not be very expensive. If the hash coder costs more than the ALU, the price would be excessive.

In hashing applications, the number of keys is usually much larger than the number of buckets in the hash table. Accordingly, the hash coder will certainly map several keys into the same bucket. If the keys are uniformly distributed into the buckets, the system performs best. If the hash coder performs poorly at the distribution of keys, the hash-based join may run slow because more disk accesses can be required. Therefore, distribution performance should be considered as a high priority. However, complicated and tediously long computations including multiplication and/or division do not guarantee a distribution of keys better than short and effective calculations.

To summarize, the requirements for the effective database hash coder should have the following attributes:

1. competitive distribution performance, compared with the current well-known hash functions (e.g., division and multiplicative methods),

2. extremely fast hash address calculation, (i.e., only few clock cycles)

3. low cost for implementation in a hardware hash coder.

In Chapter 3, various hash functions including four new ones such as Maurer's shift-fold-loading, Berkovich's Hu-Tucker code, and the author's various versions of fold-shifting and mapping are surveyed. Descriptions of hash functions are presented, along with a distribution, speed, and cost analysis, which leads to the conclusion that the new mapping hash method is a reasonable choice for the relational database hash coder when it is implemented in the hardware.

# CHAPTER 2

# MAJOR METHODS OF IMPLEMENTING THE RELATIONAL JOIN

In this chapter, the principal concepts and advantages of a database management system are discussed, and the terms of databases are defined. At this point, the discussion concentrates mainly on the relational database model, and its operators and corresponding examples are illustrated. The advantages and disadvantages of a relational database are discussed in order to highlight the possible improvements for the relational database that can be achieved by means of special hardware aids. The three major join algorithms are illustrated, and three hash-based join algorithms are discussed in detail.

## 2.1 Terms and Concepts of a Database

### 2.1.1 What is a Database?

In general terms, a database keeps data in one or more locations; therefore, a variety of program applications can access and maintain the information. A database is a collection of interrelated data that are structurally stored. Moreover, a database management system is a database capable of a set of applications such as adding new data and modify-

ing and retrieving existing data.

### 2.1.2 Advantages of a Database

According to Date <DATE1> and Martin <MART1>, an organized database offers the following major advantages. First, it reduces the amount of redundant data, thus saving storage space. Second, collected data can be shared by new and existing applications that operate on the database. Third, the database will have reduced inconsistency by virtue of allowing only one entry for each data item. Inconsistency occurs when there is duplication of an entry and the two entries are not manipulated in the same way which causes the two entries to differ from one another. Fourth, database organization increases clarity and ease of use and decreases apparent complexity in using data. Fifth, privacy is provided by security restrictions, and the protection from loss or damage is thus enhanced. Sixth, it is possible to detect inaccurate data through integrity control. Seventh, physical and logical data independence can be achieved, ensuring that the hardware storage structure or access strategy can be changed without rewriting the application program. Achieving logical data independence implies that the overall logical structures which are expanded by adding new data items do not affect the application programs. All of these features are important objectives. Finally, database organ-

ization can make application development faster, cheaper, easier, and more flexible. Some of these advantages could also be design objectives for a database.

### 2.1.3 Entities, Attributes, and Keys

To understand data structures used to implement a physical database, certain terms must be defined. The term entity means the stored information within a data item. An entity is a distinguishable object, such as a student, a classroom, or a course. A group of similar entities, such as all students, all classrooms, or all courses, is referred to as an entity set.

Entities also contain components which are referred to as attributes. An entity (e.g., student) might include a name, student identification number, grade, and major as attributes. The attributes of similar entities are then grouped into a set of attributes. For example, {name, grade} may be considered a set of attributes.

An attribute or set of attributes whose values uniquely identify each entity within an entity set is called a key for that entity set. For instance, an entity set that includes only students from one school could use the student identification number attribute as a key for that entity set.

### 2.1.4 Relational Data Model

The relational data model <ULLM1, DATE1>, consists of relations (tables or files). The members (or rows) of a relation are generally called tuples. The columns of a relation are referred to as attributes. The theoretical term domain represents the set of values from which the actual values appearing in a given column are drawn. The step-by-step normalization process <CODD1> re-forms the tables so that each attribute value in each tuple is nondecomposable (or atomic). In other words, at every row and column in the table there always exists only one value, not a set of values. Such a relation is said to be normalized. Operations, such as join, selection, and projection, are frequently used in relational operations. Several relational database operations will be explained in section 2.1.5.

The relational data model is chosen for this research, since it has many advantages over other data models, as shown in section 2.1.6; moreover, high performance database machines can be developed using this model. As a matter of fact, most existing database computers are designed to support the relational data model.

### 2.1.5 A Set of Relational Operators and Examples

This section will discuss the background of relational database operations with illustrative examples (Figure 2-1).

Relation R1

| A | B | C |
|---|---|---|
| d | e | f |
| b | d | g |
| a | d | c |

Relation R2

| D | E | F |
|---|---|---|
| a | d | c |
| d | g | a |

<Ex 1> PROJECT R2 (D, F)

| D | F |
|---|---|
| a | c |
| d | a |

<Ex 2> SELECT R1 (B = d)

| A | B | C |
|---|---|---|
| b | d | g |
| a | d | c |

<Ex 3> JOIN R1, R2 (R1.B = R2.E)

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| b | d | g | a | d | c |
| a | d | c | a | d | c |

Figure 2-1. Examples of Relational Operations

<Ex 4> UNION R1, R2

| | | |
|---|---|---|
| d | e | f |
| b | d | g |
| a | d | c |
| d | g | a |

<Ex 5> INTERSECT R1, R2

| | | |
|---|---|---|
| a | d | c |

<Ex 6> DIFFERENCE R1, R2

| | | |
|---|---|---|
| d | e | f |
| b | d | g |

<Ex 7> CARTESIAN_PRODUCT R1, R2

| A | B | C | D | E | F |
|---|---|---|---|---|---|
| d | e | f | a | d | c |
| d | e | f | d | g | a |
| b | d | g | a | d | c |
| b | d | g | d | g | a |
| a | d | c | a | d | c |
| a | d | c | d | g | a |

Figure 2-1. Continued

PROJECT Relation (Attribute List)

The projection operation involves selecting from each tuple in the relation only those attributes included in the Attribute List and then eliminating duplicate tuples in the resulting relation.

SELECT Relation (Selection Condition)

Selection Condition: a Boolean expression defined over the columns of the relation. The expression can consist of membership tests on the value of a column in the relation and of comparison tests (=, <>, <, >, <=, >=) between columns and constants or computed values.

The result of SELECTing from a relation is a new relation consisting only of the rows from the original relation that satisfy the selection condition.

JOIN Source_Relation Target_Relation (Join Condition)

Join Condition: a Boolean expression of comparisons (=, <>, <, >, <=, >=) between columns in the source relation and the target relation.

The result of JOINing two relations is a new relation consisting of all attribute headers from the two joined relations and tuples formed by concatenating tuples from the source relation with tuples from the target relation, where the join attribute in the tuples satisfies the join condition. In equi-join, the expression of comparison is limited to equal (=) only.

UNION Relation1, Relation2 (Attribute-pair List)

Attribute-pair List: a list of pairs consisting of an

attribute name from Relation1 and an attribute name from Relation2.

The union of Relation1 and Relation2 is the relation of tuples that are in Relation1 or Relation2, or both, with any duplication removed and the attributes ordered in the sequence given in the Attribute-pair List.

INTERSECT Relation1, Relation2 (Attribute-pair List)

The result of INTERSECTing two relations is a new relation consisting of only those rows appearing in both relations.

DIFFERENCE Relation1, Relation2 (Attribute-pair List)

The result of DIFFERENCing two relations is a new relation consisting of tuples that are in Relation1 but not in Relation2.

CARTESIAN_PRODUCT Relation1, Relation2

The Cartesian Product of Relation1 and Relation2 is the relation of tuples that consist of combined attributes from Relation1 and Relation2.

### 2.1.6 Advantages of the Relational Data Model

The relational database model has many advantages over other models <CODD1, MART1>. First, the relational model provides the easiest and simplest way of representing data

to database users by showing two-dimensional tables. Second, relational database operations, such as project and join, provide a great amount of flexibility by cutting and pasting relations as users need them. Third, relations are flat files; therefore, they are much less complex than tree and network structures with pointers and linkages. Also, the normalization offers more simplification of data storage; therefore, it is easier to implement the relational data model. Fourth, the relational model can easily achieve more data independence than others whenever tuples are added or deleted. If the database is in a normalized form, it can grow or shrink without requiring existing application programs to change. Thus, the cost of maintaining those programs can be reduced. Fifth, security controls can be easily implemented by moving sensitive attributes into a separate relation with its own authorization controls. Sixth, the relational algebra or relational calculus can be accurately applied to the manipulation of relations. However, logical data representations with directed links often mislead users. Finally, the relational model can be effectively implemented with current VLSI technologies such as associative memories, intelligent secondary storage devices, and multiprocessors.

### 2.1.7 Disadvantages of the Relational Data Model and Solution Approach

One disadvantage of the relational data model is often attributed to machine performance. With conventional processors, the join operation is most likely to take substantial machine time. When the sizes of relations are large (i.e., millions of tuples), performance may be drastically reduced due to the conventional way of performing a join during which machine time is proportional to the square of input size. Physical structure of the data and techniques can be selected to increase the speed of time-consuming and frequently used operations such as the join. Therefore, the performance of a relational database system is largely dependent on the chosen physical techniques.

## 2.2 More about the Join Relational Database Operation

The join operation described previously has been called "explicit join," in contrast to "implicit join" <SU1> 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. As a consequence, 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. Semi-joins <BERN1> are generally a more efficient way of performing joins. If S <A # B> T is a join of relation S and T over join attribute A and B, respectively, with respect to a # comparison operator such as =, then it can be equivalently processed by a semi-join as follows;

S <A # B> T = ((T <B>) <<B # A>> S) <A # B> T

the double parentheses signify semi-join.

According to the previous expression, join is actually equivalent to the projection of the T relation on the join attribute B, the semi-join of the resulting relation with the S relation over the join attribute A, and the join of the second resulting relation with the T relation. The join operation is thus divided into the primitive operations such as projection and semi-join so that the project and semi-join operations provide reduction at an earlier stage.

This section explains the three major methods of implementing the join: nested-loop, sort-merge, and hash join. Because the new join method will be a hash-based join, emphasis is placed on the following hash join methods: simple, GRACE, and Hybrid hash.

## 2.3 The Nested-Loop Join Method

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

Considering the actual implementation of the nested-loop join method, pages of tuples from 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 the outer relation S has been joined with K pages (where K is a system variable such as the number of tracks in a cylinder) of the inner relation T, another set of K pages of T are read and compared with the same page of the outer relation S. The join attributes of the outer relation S are then compared with the join attributes of every tuple in the K pages of the inner relation T. Concatenations of tuples are formed in an output buffer if they satisfy the join condition. Whenever the output buffer is full, the partial outcome of the resulting relation R is recorded on the secondary stor-

age. This process continues until all the pages of the inner relation T have been joined with the page of the outer relation S. At this time, the next page of the outer relation S is read from the secondary storage, and the process of joining one page of the outer relation S with all pages of the inner relation T is repeated. The algorithm terminates when the last page of the outer relation S has been processed.

## 2.4 The Sort-Merge Join Method

Each of the source (S) and target (T) relations is retrieved from secondary storage, and their tuples are sorted over one or more join attributes in subsequent phases using one of many sorting algorithms (e.g., n-way merge). After the completion of the sorting operation, the two sorted streams of tuples are merged together. During the merge operation, if a tuple of the 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 was 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 <VALD1, BITT1>. The partial sort-merge join algorithm iteratively merges b runs of b**(i-1) pages into a sorted run of b**i pages starting from i=1 where a run is defined as an ordered sequence of elements. The value of i is incremented for each successive iteration until all elements have been processed and included in the b runs. When a page is read for the first time, it is sorted using an internal sort algorithm. The merge of b runs is completed by successively reading one necessary page or each run into b input buffer and moving ordered tuples into the output buffer which is emptied to the cache memory when the output buffer is full. The sorting process is applied on both source and target relations; the two sorted relations are then joined by merging them.

## 2.5 The 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 paper <DEWI3>, this section illustrates how the three major hash join methods, namely the simple hash join, GRACE hash join, and Hybrid hash join overcome the limitation of real memory size.

### 2.5.1 General Approach of Hash Join

In the straightforward hash join algorithm, the source and target relations, which we call 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> also take into account the possibility that a hash table for the source relation will not fit into main memory.

### 2.5.2 Simple Hash Join

If a hash table containing all of a 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 adequate, the simple hash join scans the (smaller) relation 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, the tuple is inserted into the addressed bucket of the hash table in main memory; otherwise, the tuple is written to a source file on disk. After completing this operation, the algorithm 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, the algorithm compares 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 written to disk; otherwise, the T tuple is written to a target file on disk. Finally, the algorithm replaces relations S and T by the source and target files on the disk respectively and chooses another range; it then repeats the above steps until there are no more tuples in the source file.

### 2.5.3 GRACE Hash Join

The GRACE hash join is characterized by a complete separation of the partition phase and the sorting phase. The partition phase chooses a hash function h and creates only as many buckets from the 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 tuple of S is scanned, hashed, and placed in the corresponding output buffer. When an output buffer is filled, the accumulated tuples in it are written to a file referred to as subset Si on disk. After all tuples of S have been completely processed, all output buffers are flushed to the disk. Then, the target relation for T is processed in the same way as it was for S. If the number of buckets equals N, the N subset files for S and the N subset files for T are written onto the disk.

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

### 2.5.4 Hybrid Hash Join

The final type, the Hybrid hash join algorithm, does both partitioning and hashing on the first pass over the

source (S) and target (T) relations. All partitioning is completely finished in the first stage. While this feature is similar to the GRACE algorithm, it is different from the simple hash join algorithm. However, this feature differs 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 reserving only the blocks necessary to partition S into buckets that can fit in real memory.

The first step of the Hybrid algorithm is to choose a hash function and set up both buffers and a hash table by allocating necessary blocks of memory. After the i-th output buffer block is assigned to Si, for i = 1,...,n, each tuple of S is scanned and hashed with the chosen hash function. If a tuple has the address 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 the target relation T is scanned and hashed with the same hash function used for S. If the tuple has the address T0, the hash table is searched for a match. If there is a matched source tuple, the resulting tuple is stored. Otherwise, the target tuple is discarded. If the tuple does not have the address T0, it belongs to Ti for

some i > 0, so the tuple is moved to the i-th output buffer block. After this step, the subset files S1, ..., Sn and T1, ..., Tn are written 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.

## 2.6 Discussion

When the number of tuples in the source relation (the smaller relation) is S, the number of tuples in the target relation (the larger relation) is T, and the number of tuples in the resulting relation is R, then the time complexity of nested-loop join algorithm is O(S*T), and the time complexity of the sort-merge algorithm is O((S+T) log (S+T)), which is O(N log N). Considering only the time complexities, the sort-merge join may outperform the nested-loop join. However, as mentioned before, if a suitable index exists, a nested-loop can be a choice as well according to Blasgen's analysis <BLAS1>.

For parallel join operations, Bitton and his research colleagues analyzed parallel sort-merge and parallel nested-loop join algorithms and concluded that, when the sizes of the two relations to be joined are approximately the same,

the parallel sort-merge algorithm is superior to the parallel nested-loop algorithm <BITT1>. Furthermore, the authors added that when one relation is larger than the other, the parallel nested-loop algorithm is faster.

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

Considering actual performances, it is hard to rely only on asymptotic complexity analysis to measure the speed performance because of I/O time, communication overhead, and a number of accesses to the secondary storage are also needed for a more accurate analysis.

Shapiro <SHAP1> and Schneider <SCHN1> analyzed simple, GRACE, and Hybrid hash join algorithms. They 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-based join algorithm requires a large main memory; however, when sufficient main memory is affordable, the hash-based join has the greatest advantage.

# CHAPTER 3

## SURVEY OF HASH FUNCTIONS FOR IMPLEMENTING AN EFFECTIVE HASH CODER

During the 1970's and 1980's, implementing a powerful sorter in computer hardware was a challenging topic. 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, that is repeatedly used, takes a long process time, it is important to think about implementing a piece of hardware to do the job faster. Since computer hardware has become much less expensive than in the past, many of these ideas regarding implementing new pieces of hardware have become more feasible.

These days, distributive sorting by a hash function is popularly used in many applications; therefore, there has been an increasing demand for an effective hash coder. In these database applications, an effective hash coder with an efficient hash function is essential to increase the speed of the hash-based join operation. Since the hash coder is the major component in the database filter coprocessor, it should be effectively implemented in hardware in order to hash database tuples at an increased speed.

This chapter will discuss the major objectives in

designing a hash coder for use with a join operation. In the second section of this chapter, the experimental environment, including the encoding scheme, data sets, the measurements of distribution, speed, and cost are described. The third section describes four new hash functions and current hash functions. In addition, an analysis of the critical aspects of each hash function is provided in the fourth section of this chapter. Finally, a new hash function, named mapping hash, has been chosen as the hash coder which is implemented in hardware for relational database operations.

## 3.1 Objectives in Designing a Hash Coder for Use with a Join Operation

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

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

The first requirement is crucially important in this database application since the number of join attributes the hash coder has to transform into hash addresses may be large. The hash coder constitutes the major component of a database filter that strains out irrelevant tuples transferred from the secondary storage devices. Within the fil-

ter device, the hash address calculation per each join attribute often is a main cause of time consumption.

Knuth's second requirement for minimizing collisions implies that a good hash function should provide a good distribution performance. 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 used to write tuples in a bucket to a subset file can be reduced. Since no hash function can distribute an equal amount of keys in each bucket, it becomes necessary to compare the distribution performance of any new hash function with currently accepted hash functions such as division and multiplicative hash methods. Knuth says that even though many hash methods have been suggested, none has proven to be superior to the simple division and multiplicative methods <KNUT1>. His conclusion, based on Lum and his colleagues' experimental results <LUM1>, was generally accepted as true until very recently.

According to this survey of hash functions, distribution performance of some hash functions might show a data dependency problem. In other words, when keys are similar, a data dependent hash function has a larger chance of collision occurring. For example, by using the division method algorithm, keys such as 'contract1,' 'contract2,' 'contract3' probably will be distributed into buckets next to one another. This phenomenon can be described as data clus-

tering due to the data dependency of the division method. To detect the data dependency, each hash method should be applied both to the data set, which contains many similar keys, and to the data set, which contains many random keys, while the number of buckets should be sufficient, i.e., at least several hundred buckets. If there is a distinguishable difference in the two distribution performances of a hash method, the hash method may have a data dependency problem.

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

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

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

1. Extremely fast hash address calculation
   (i.e., a few clock cycles)
2. Relatively good distribution performance
3. Data-independent distribution
4. Low cost in implementation

Since most of the currently well known hash functions receive an input number and produce another number representing a bucket, an encoding scheme is required in order to convert the character string of a key to an input number. Therefore, their hashing techniques are reminiscent of pseudo-random number generating techniques <KNUT2> which receive an input number and produce another number. For a fast address calculation, unnecessary serial encoding schemes have to be avoided if it is found that the process of encoding one key to another short form slows the computation. Nonetheless, when a key is long and encoding is unavoidable, the hardware encoding scheme, such as arrays of exclusive-OR gates that look like a downward binary tree, can be used.

Another factor to consider is that if the hash address is represented with K bits, the number of buckets in a hash table is 2**K. Therefore, if the circuitry for a hash function involves a sequential network, requiring tedious serial computations, the first requirement cannot be satisfied.

Moreover, if the sequential network for a hash function, has a series of complex operations and is replaced by a combinational network, then the cost of the hardware hash coder may be increased. However, if the algorithm of a new hash function can be processed using only a combinational circuit, then a hash address can be generated within a few clock cycles.

Among the current hash functions, folding and digit analysis are the hash methods that inherently fit well in a combinational network. Most other hash functions require sequential circuits by nature. However, Lum states that both folding and digit analysis are erratic <LUM1>. Maurer <MAUR1> and Knott <KNOT1> suggest that the folding method should be combined with a shift operation(s) in order to improve the distribution performance. However, this fold-shifting hash method may take more time than folding alone. The digit analysis hash method, on the other hand, is in a different category which requires that the specific data set should be analyzed beforehand to select digits; therefore, further discussion of this method is not necessary at this moment. Maurer stated that the fold-shifting method is probably the fastest, followed by the division method. The principle behind all these fast hashing techniques is that folding using exclusive-OR, shift, and negation are fast and useful operations for generating hash addresses.

In section 3.3, four new hash methods are introduced.

The mapping hash method is the one developed by the author, and two other hash methods, fold-shift-loading and Hu-Tucker code methods, were developed by professors Maurer and Berkovich at the George Washington University, respectively. In this section, several different versions of the fold-shifting method are designed by the author and introduced. In the end, this dissertation will show how the mapping hash method satisfies all the requirements of a good hash coder that have been illustrated above.

## 3.2 Experimental Environment

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

This experiment assumes that records (or tuples) with keys (or join attributes) are moved from a sequential file in the secondary storage to a buffer memory: the keys are hashed by a chosen hash function, and each record is stored into a corresponding bucket in a hash table in the main memory. When the bucket is filled, the accumulated records are written to the corresponding subset file in secondary storage. In this environment, lookup time--the time to look up a record on the main memory--is relatively fast compared to

the hash address calculation time. Under these circumstances, the speed performance of a hash function is an important criterion in comparing various hash methods.

Three kinds of data sets are used to compare the performance of hash functions. Keys in these three data sets consist of 16 identifiable characters; they are left justified and are space character filled. In this experiment for a survey of hash functions, a key consists of 16 ASCII characters, which is an acceptable size for the keys used in most of the database applications. The first data set includes 1,024 generally or arbitrarily chosen persons' names with 16 characters. In this data set, there are dozens of groups of people having the same last name. The second data set contains 1,024 persons' names, randomly chosen from the phone book, depending on the row, column, and page number, generated by a pseudo-random number generating function. The third data set has 1,024 numbers with 16 numeric characters, which are generated by the same function. In this last data set, every key consists of only numeric characters such as '0,' '1,' ..., '9.'

Each character in the data sets is internally represented by its corresponding ASCII code. Although this code uses seven bits, most of the computers have eight bits for a byte or a character. Therefore, the left most bit always has zero value in the ASCII code. It is assumed that 16 characters in the ASCII code are initially stored in a four-word,

or 16-byte, key register. If the ASCII code character string is considered as a number, it may be too large for some hash functions to calculate. Therefore, in this survey, an encoding scheme is used for hash functions such as algebraic coding, digit analysis, division, folding, midsquare, multiplicative, radix, random, and the author's fold-shifting(FS). On the other hand, hash functions such as Maurer's shift-fold-loading, Berkovich's Hu-Tucker code, the author's mapping, and Pearson's table indexing do not use encoding schemes.

There are many encoding schemes that one can use with a hash function. If a key is encoded into one word, most of the existing hashing function can be directly applied. As Maurer suggests <MAUR1>, if keys are longer than one computer word, each word in a key can be folded to the next one consecutively, taking the exclusive-OR. 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 (last) word of the key in order to produce the highest bit of the encoded word. The same process is applied to all other bits in parallel because the exclusive-OR operation is taken word by word. Because this encoding scheme is fast and easily implemented in both hardware and software,

it merits attention.

Since this treatise focuses on a fast hardware oriented hash function, calculated hash addresses should be represented with the values of the address bits--8 address bits in this case. The number of buckets in the hash table is 256, 2 to the power of 8. Considering that a 16-character key is composed of 128 bits, i.e., 16*8 bits, which are input to a hash function to produce eight bits for a hash address as an output, the encoding scheme that uses folding with exclusive-OR operations can be efficiently and easily implemented. The choice of 256 for the number of buckets in a hash table provides fairness for both hardware- and software-oriented hash functions as indicated by a comparative analysis of their performances.

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 an informed 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 the function is implemented in both hardware and software. As the barometer of distribution performance, mean square deviation is selected. Each hash method is executed on the three data sets to produce mean square deviations to show its distribution performance. The smaller the mean square devia-

tion, the better the distribution and the fewer incidences of collision. Since the number of buckets in the hash table is 256 (2**8), and since 1024 keys are hashed with a uniform distribution, each bucket will contain four tuples--the mean. The formula of the mean square deviation is:

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

Ni : the number of tuples in bucket i

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

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

It should be noted that the executional speed of most hash functions will be faster when the hash coder is implemented in hardware as compared to software. Thus, two speed performances should be determined: one for a software implementation and the other for a hardware implementation.

First, when a hash function is implemented in software, execution time in clock cycles can be calculated by hand. The 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 is analyzed as a whole. Because the host processor of the HIMOD database computer uses an MC68030 microprocessor, the actual instruction-cache case execution time 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 overlapped with the instruction-cache case time for the operation. The overlap time should be subtracted from the address calculation time for the entire sequence as shown in formula in the MC68030 User's Manual <MOTO1>. The formula is used in every calculation of the whole execution time that is taken to produce a hash address.

Second, it should be noted that when a hash function is implemented in hardware, the execution time in the clock cycle is calculated for each hash function based on the following considerations: the hardware hash coder may simply consist 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 a maximum nine 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 a number of gate levels (L) less than or equal to 5 (i.e., L < clock pulse width/gate delay = 50/9), then the signals can travel from the inputs

of the circuit to its outputs within a clock cycle.

Third, it should be noted that the cost of a hardware implemented 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. The gates used for the key register which is provided to all hash methods are not included in the number of gates used in the hash coder. If any other device or local memory is used, it is specified in addition to the number of gates by using a postfix mark.

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

made thoughtfully. For that reason, the gates of these options also are reflected in the costs of a hash coder in order to help a computer designer make the best decision.

According to this survey of hash functions, key-to-address transformation methods are evaluated without weighing other factors such as overflow storage or handling schemes, loading factor (the ratio of the number of records to the number of record slots which are units of storage space that can hold one record), bucket size (the number of records that can be accommodated in a location calculated from a transformation), and insertions and deletions, owing to the environment of the database application.

## 3.3 Description of New and Current Hash Functions

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

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

Initially an input key exists in both shift and key reg-

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

The algorithm is:

```
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))
```

where the statement 'N := N EX-OR (ROT-R(N, K))' assigns the resulting value from exclusive-OR of both intermittent value (N) and K bits rotated value of N back to the intermittent value.

As specified in the algorithm in the second rotation, all the bits in the shift register are rotated three bits right, and exclusive-ORing and loading follows by the same method as described above. Then the algorithm rotates seven bits right, while performing the same exclusive-ORing and loading once again. It then rotates another 15 bits right and repeats the process. After that, the same process for

31, 63, and 127 bits is duplicated in order.

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

$$K1 = 2^{1} - 1 = 1$$
$$K2 = 2^{2} - 1 = 3$$
$$K3 = 2^{3} - 1 = 7$$
$$K4 = 2^{4} - 1 = 15$$
$$K5 = 2^{5} - 1 = 31$$
$$K6 = 2^{6} - 1 = 63$$
$$K7 = 2^{7} - 1 = 127 < 128 \text{ (N : Number of bits in a key)}$$

If there are N bits in a key, log N numbers of shift, exclusive-OR and load operations are required, since Ki = 2**i - 1 < N (i >= 1). Therefore, 1 <= i <= log N.

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

In Berkovich's Hu-Tucker code hash method, the Hu-Tucker variable length code <KNUT1>, as shown in Figure 3-1, is used. Converting each character in a key to its corresponding Hu-Tucker code and storing the binary string of the code for each character, the Hu-Tucker code string for the whole key is accumulatively created, character by character. For example, the Hu-Tucker coded value of the key 'ABC' is '0010001100001101.' In the conversion process, the string size of a code for each character must be added to provide

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

| | | | |
|---|---|---|---|
| SPACE | : 000 | n, N | : 1010 |
| a, A | : 0010 | o, O | : 1011 |
| b, B | : 001100 | p, P | : 110000 |
| c, C | : 001101 | q, Q | : 110001 |
| d, D | : 00111 | r, R | : 11001 |
| e, E | : 010 | s, S | : 1101 |
| f, F | : 01100 | t, T | : 1110 |
| g, G | : 01101 | u, U | : 111100 |
| h, H | : 0111 | v, V | : 111101 |
| i, I | : 1000 | w, W | : 111110 |
| j, J | : 1001000 | x, X | : 11111100 |
| k, K | : 1001001 | y, Y | : 11111101 |
| l, L | : 100101 | z, Z | : 1111111 |
| m, M | : 10011 | | |

Figure 3-1. Hu-Tucker Codes <KNUT1>

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

### 3.3.3 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 arbitrarily chosen prime number (or a randomly chosen number) in parallel, and then folds these prime numbers using arrays of exclusive-OR gates to produce a number in binary form and, once again, in a parallel manner. Then the function extracts K bits from the binary number in order to produce a hash address for the hash table of 2**K buckets.

In this hash method, the nature of prime numbers that have inherent irregularity are used to randomize each ASCII character. For instance, the ASCII codes of the characters 'A' and 'B' are different in only one out of eight bits. However, this kind of similarity in ASCII code for both alphabetic and numeric characters does not help randomizing the value for a hash address. Therefore, whenever a character is converted into a prime number, the similarity between ASCII codes of the characters in a key disappears. Moreover, when these prime numbers are compressed or folded into a number, this number must be randomized sufficiently. Through the folding process of prime numbers, the different values in one character position of two similar keys will produce two totally irrelevant hash values. In other words, the prime number for the different character in the second key affects all bits of the hashed value of a key due to the exclusive-OR operation. Therefore, in the final exclusive-OR process, the whole bits of hash value are completely hashed

by the character.

One of the characteristics of this hash method occurs when this hash function is implemented in hardware: a hash address can be calculated within a few machine cycles by means of parallel processing. The mapping hash coder requires hardware components, for example, sixteen 64*16 bits ROMs (one ROM for each character) or RAMs and eight exclusive-OR or EX-OR modules (120 exclusive-OR gates in total) as shown in Figures 3-2 and 3-3.

In each ROM, 64 (2**6) the arbitrarily selected prime numbers are stored. (Each ROM may contain 128, 32, or 16 prime numbers if a user chooses 128*16, 32*16, or 16*16 ROM respectively.) A set of all 16 ROMs is included in the hardware mapping hash coder. The contents of all 16 ROMs are different. In this hash coder, only the least significant six bits of an ASCII character are used as an input address to the corresponding ROM, since the 7th bit is always '1' for alphabets and always '0' for numerics, and the 8th bit is not used in the ASCII code. Therefore, each ROM contains only 64 words of prime numbers, which are addressed by the least significant six bits of the ASCII code for the corresponding characters in the key. When the least significant six bits of each ASCII character code select a prime number in a corresponding ROM, the 16 prime numbers for 16 corresponding characters in a key are exclusive-ORed together to produce a hash address in parallel.

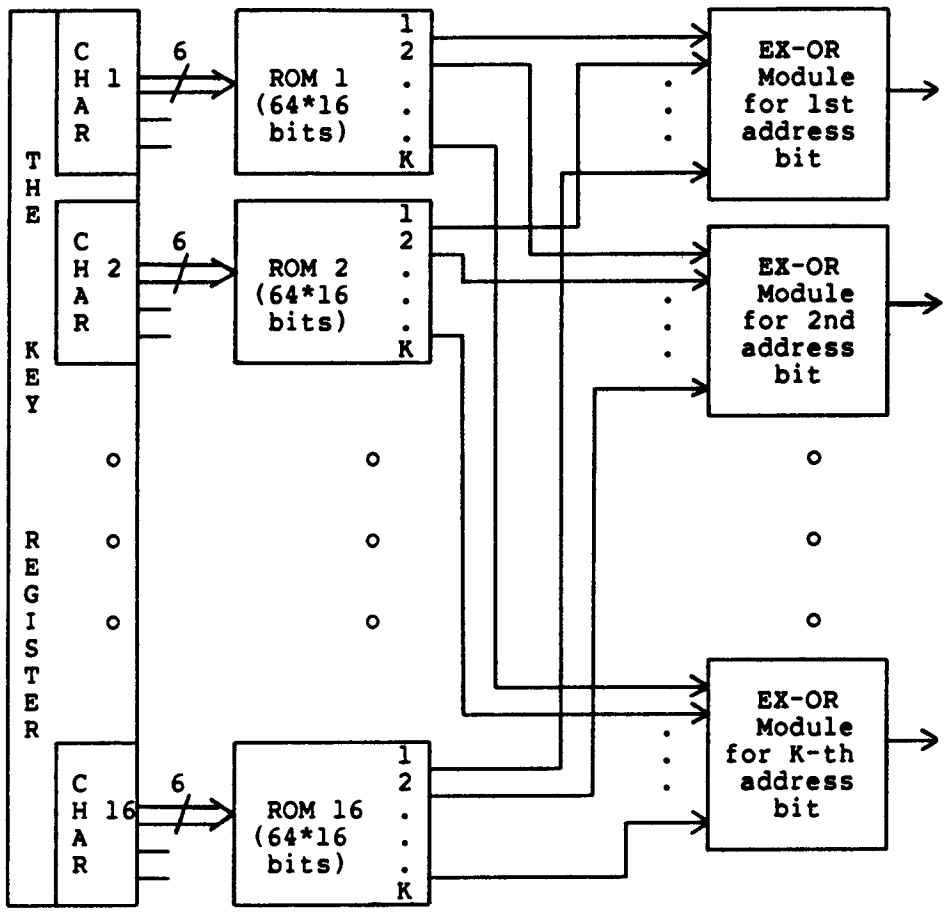


Figure 3-2. The Hardware Mapping Hash Coder

As shown in Figure 3-2, the first bits of the 16 prime numbers are exclusive-ORed together to generate the first bit of a hash address. Simultaneously, the second bits of the 16 prime numbers are exclusive-ORed together producing the second bit of the resulting hash address. All other bits

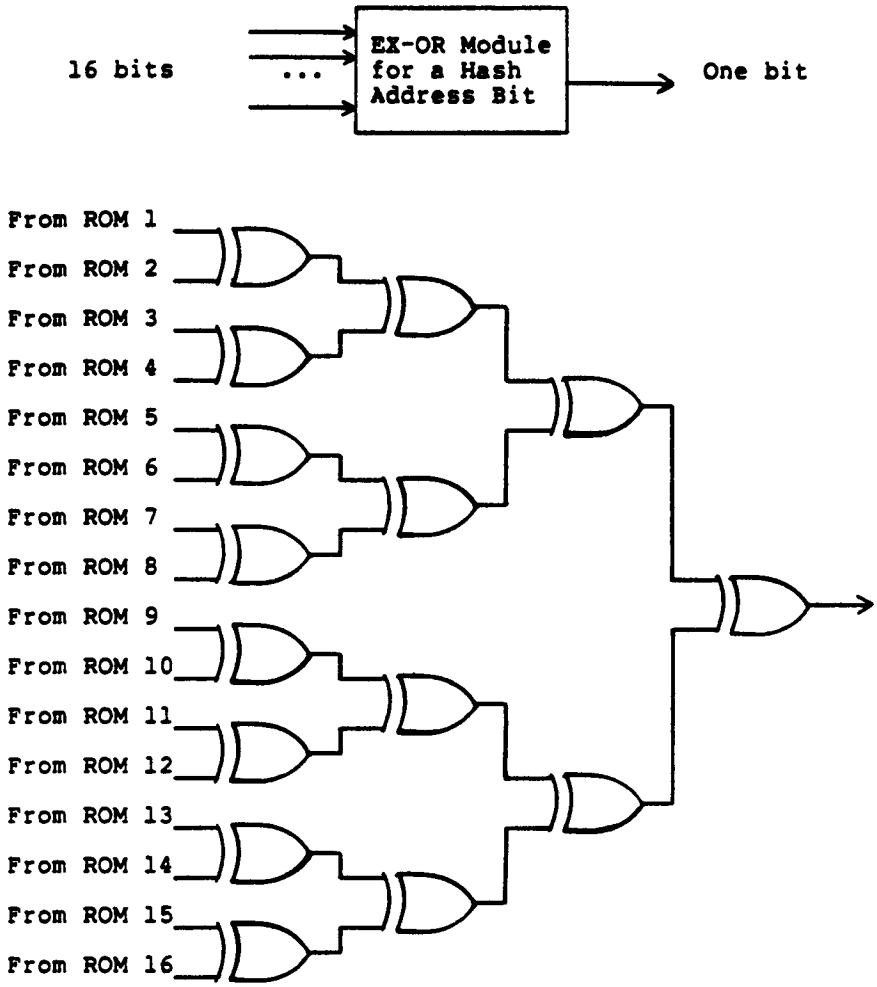


Figure 3-3. EX-OR Module for a Hash Address Bit

of a hash address also are constructed at the same time. The circuit of EX-OR Module for each bit shown in Figure 3-2 is represented in Figure 3-3.

As shown in Figure 3-2, while the first bit of a hash address is being calculated, the other bits are also being

computed through the four levels of exclusive-OR gates in parallel. 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. These operations also are time-saving operations.

The conversion of an ASCII character to a prime number is a useful aid in randomizing the value of bits. It is, however, necessary to be cautious about designating the least significant bit of every prime number '1,' since prime numbers are odd numbers. Consequently, the resulting least significant bit, designated '0,' should be excluded in forming a hash address. It may be possible to remedy this process by adding one to the prime numbers of all even number addressed words in every ROM. If such additions are made, then the resulting first bit might be adequately randomized in order to be included in composing the bits for a hash address.

By using available instructions, this hash method also can be implemented in software. The algorithm of this hash method in a Pascal-like notation is shown in Figure 3-4. The programming language of Pascal provides an ORD function which converts a character to a corresponding ASCII integer number. In the algorithm, only the six least significant bits of the ASCII numbers are used as indices to the table containing 64 (= 2**6) prime numbers. The following state-

ment in the algorithm, "Temp := EX_OR (Prime_Table (Index), Temp);" performs exactly the same function that the hardware implemented mapping hash method does. The validity of this assertion will be further demonstrated in this section.

```
const
     MAX_NO_CHARS_IN_KEY = 16; {number of chars in a key}
     MAX_NO_BUCKETS = 256; {no. of buckets in hash table}
     NO_PRIMES_IN_ROM = 64;

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

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


function Mapping_Hash (Key : Key_Array_Type) : integer;

var
     Temp, Char_No, Index : integer;

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

Figure 3-4. Mapping Hash Algorithm

If the Exclusive-OR (EX_OR) function is explained in

high level terms, it receives two integer numbers to be exclusive-ORed, which then converts them to two strings of 1's and 0's, and takes the exclusive-OR on the bits of the same position in the two strings. Then the mapping hash method converts the resulting binary string back to integer output to be sent to the calling program. In hardware implementation of this mapping hash method, the exclusive-OR operation is more valuable than the addition operation since the exclusive-OR operation does not generate a carry-out bit. Should anyone implement this hash function in a high level language while disregarding speed, the following statement can be used: "Temp := Prime_Table(Char_No,Index) + Temp;" in place of the statement: "Temp := EX_OR (Prime_Table(Char_No,Index), Temp);" This hash method is referred to as the author's additive mapping hash method, and gives as good a distribution performance as the mapping hash method, as shown in section 3.4.

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

The assertion that parallel processing with exclusive-OR gates, as shown in Figures 3-2 and 3-3, has the same

effect with serial processing as it does with exclusive-OR operations as shown in Figure 3-4 remains to be proved. In other words, taking serial exclusive-ORs on 16 prime numbers ensures the same result of collecting all bits in the same bit position of 16 prime numbers, by taking exclusive-ORs in parallel passing through the EX_OR module, as shown in Figure 3-3, and then producing resulting bits from EX_OR module for a hashed value. It is easier to understand this process if one considers how the first resulting bits in the serial and the parallel processing cases are produced, as well as how their results are the same. If '⊕' is understood to represent exclusive-OR operation on two input bits, and X1 is the first bit of the first prime number, then X2 is the first bit of the second prime number, and so on. As a result, Xi is the first bit of the i-th prime number. The assertion to be proved can be expressed with the following equation:

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

is equal to (=)

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

The left-hand side of the equation represents how the parallel exclusive-ORs on the first bits are taken from the 16 prime numbers. The right-hand side of the equation repre-

sents how the serial exclusive-ORs on the first bits are taken from the 16 prime numbers. By using the associative law of exclusive-OR, such that (X⊕Y)⊕Z = X⊕(Y⊕Z) = X⊕Y⊕Z, one can simplify the right-hand side as follows:

R.H.S. =

X1⊕X2⊕X3⊕X4⊕X5⊕X6⊕X7⊕X8⊕X9⊕X10⊕X11⊕X12⊕X13⊕X14⊕X15⊕X16

Thus, the equation to be proved becomes:

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

is equal to (=)

X1⊕X2⊕X3⊕X4⊕X5⊕X6⊕X7⊕X8⊕X9⊕X10⊕X11⊕X12⊕X13⊕X14⊕X15⊕X16

Once again, the associative law of exclusive-OR is used in order to accomplish the following steps:

L.H.S. = (((X1⊕X2)⊕(X3⊕X4))⊕((X5⊕X6)⊕(X7⊕X8)))⊕
(((X9⊕X10)⊕(X11⊕X12))⊕((X13⊕X14)⊕(X15⊕X16)))

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

by the associative law (X⊕Y)⊕Z = X⊕Y⊕Z

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

by the associative law X⊕(Y⊕Z) = X⊕Y⊕Z

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

by several uses of the associative law (X⊕Y)⊕Z = X⊕Y⊕Z

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

by several uses of the associative law X⊕(Y⊕Z) = X⊕Y⊕Z

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

by several uses of the associative law (X⊕Y)⊕Z = X⊕Y⊕Z

= X1⊕X2⊕X3⊕X4⊕X5⊕X6⊕X7⊕X8⊕
X9⊕X10⊕X11⊕X12⊕X13⊕X14⊕X15⊕X16

by several uses of the associative law (X⊕Y)⊕Z = X⊕Y⊕Z.

As this method demonstrates, the first resulting bits of parallel processing and the first resulting bits of serial processing have the same bit value. This is true of all other resulting bits of parallel and serial processing. Therefore, it is verified 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 when they functionally use the same hash address calculation method.

### 3.3.4 The Algebraic Coding Hash Method

Each digit of a key is regarded as a polynomial coefficient in the algebraic coding hash method <MAUR1, LUM2>. If a key is n digits (or bits) long, the degree of the polynomial becomes n-1. For example, the key 247935 is translated as 2X**5 + 4X**4 + 7X**3 + 9X**2 + 3X + 5 which is the polynomial of degree 5 (n=6). In order to produce a remainder, the polynomial is divided by another constant polynomial of

degree m-1 (m<=n). The coefficients of the remainder, which is the polynomial of degree p (p<=m-1), form a hash address.

This method of key transformation is organized according to the theory of error-correcting codes. Hanan and Palermo <HANA1> applied the theory of Bose-Chaudhuri codes to a hashing technique. In their method, the key and addresses are represented as polynomials such as:

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

If 'a' represents a primitive element of Galois Field GF(2**q), the one must consider the polynomial

$$g(x) = (x-a)(x-a^{**}2)\ldots(x-a^{**}(d-1)) = \sum_{i=0}^{d-1} g_i x^i,$$

where d <= D (D : distance)

The Bose-Chaudhuri theorem <HANA1> states that R(x) is the remainder of the division of K(x) by g(x); that is, given that K(x) = Q(x)g(x) + R(x), degree of R(x) < d - 1, then the minimum distance between two keys producing the same R is at least d. According to this theorem, all keys, which are at a distance of D, or less, apart, have distinct remainders--that is, distinct hash addresses. In order to implement this method, the division K(x) by g(x) can be performed by a computer or by a stage shift register.

### 3.3.5 The Digit Analysis Hash Method

The digit analysis hash method <MAUR1, LUM1> differs from all others in that it deals only with a static file where all the keys in an input file are known beforehand. Therefore, using either mean square deviation or standard deviation, the skewed distribution of each digit or bit position can be analyzed. The digits that have the most skewed distributions (larger deviations) are deleted to make the number of digits left, small enough to produce an address in the range of the hash table. This statistical analysis does not guarantee uniform distribution; however, it does provide a better chance of producing uniform spread.

### 3.3.6 The Division Hash Method

Currently, the division hash method is the method most frequently used. As far as distribution performance is concerned, it is believed that no hash function is superior to the division method. Most researchers in this field agree <BUCH1, LUM1, MAUR1, KNUT1>. This hash algorithm simply adds, or exclusive-ORs, the ordinal number of words in a key and takes the remainder, and divides the sum (the combination or the encoded key, K) by bucket size number B. The resulting remainder (h(K)) could represent any bucket number 0 through B-1.

Buchholz and Maurer suggest that the divisor should be

the largest prime number smaller than B <BUCH1, MAUR2>. Lum and his research colleagues argue that the divisor does not have to be a prime; they propose that a nonprime number with no prime factors less than 20 will work as well <LUM1>.

### 3.3.7 The Folding Hash Method

In the folding hash method <MAUR1, LUM1>, the key is partitioned into several parts; e.g., 3 partitions in the key are folded inward like folding paper. Subsequently, the bits or digits falling into the same position are exclusive-ORed (or added). The K bits in the resulting partition are then used to represent a hash address. This folding method is specifically called fold-boundary or folding at the boundaries.

In another folding method, all but the first partitions are shifted so that the least significant bit of each partition lines up with the corresponding bit of the first partition, then these partitions are folded. This method is often referred to as fold-shifting or shift folding. New versions of fold-shifting are developed and discussed in the next section.

### 3.3.8 The Fold-shifting Hash Method

As has been shown by several researchers <MAUR1, KNUT1, KNOT1, LUM1>, the fold-shifting hash method is the fastest

and most easily implemented method in hardware. In hardware implementation of the fold-shifting hash method, the original encoded keyword can be shifted, not by a shift register, but by wires which are shifted in their connection to exclusive-OR gates.

Since various fold-shifting techniques are used in cryptology <MELL1, SIEG1>, when there is an encoded one word key (or partition) there may be many ways to fold using the exclusive-OR operation. The questions about the fold-shifting method can be described as follows:

1) How many partitions have to be made on a key? (Or how many folding processes are needed?)

2) How many bits should be shifted or rotated for each partition?

In answering the above questions, it is necessary to consider how many shifted keywords are needed in folding in order to randomize the bits in the resulting word. Since the bit patterns of ASCII still affect each bit in the encoded key, it is easy to see that there are similar patterns in an encoded key. In other words, each byte in an encoded key may have a similar pattern. However, the pattern in each byte should be eliminated in the folding process. Hence, the scope of randomization is narrowed down to a byte. If the number of bits rotated is one, then eight

rotated keywords might be sufficient to randomize every bit in a byte, since eight, the number of keywords, times one, the number of bits rotated, is the number of bits in a byte. This fold-shifting process may be represented by R(0,1,2,3,4,5,6,7).

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

When the number of bits rotated is three, R(0,3,6,1) would be considered. If the number of bits rotated is four, R(0,4,1,5) can be used instead of R(0,4,0,4) or R(0,4). If five bits are rotated, R(0,5,2,7) can be used. When six bits are rotated, R(0,6,4,2) would be considered; however, it is symmetrical to R(0,2,4,6); therefore, R(0,6,4,2) would

not be selected. If seven bits are rotated, R(0,7,6,5) can be used.

These fold-shifting hash methods may require that the number of rotated keywords should be four (2**k, k=2) because two is too little and eight is too many. Interestingly, there are four bytes in an encoded keyword, and the number of rotated keywords are four. Therefore, it can be deduced that at least some portion of each byte should affect the other three bytes in the keyword. Accordingly, eight bits should be rotated right in the second keyword, 16 bits should be rotated right in the third keyword, and 24 bits should be rotated right for the fourth keyword. Thus, R(0,2,4,6) becomes FS(0, 2+8, 4+8*2, 6+8*3) or FS(0,10,20 30). By the same process, R(0,3,6,1) becomes FS(0,11,22,25), R(0,4,1,5) becomes FS(0,10,17,29), R(0,5,2,7) becomes FS(0,13,18,31), and R(0,7,6,5) becomes FS(0,15,22,29).

The selected fold-shifting hash methods to be examined are FS(0,10,20,30), FS(0,11,22,25), FS(0,10,17,29), FS(0,13,18,31), and FS(0,10,17,29). Their distribution performances are discussed in section 3.4.

### 3.3.9 The Midsquare Hash Method

In the midsquare hash method <MAUR1, LUM2>, the key is multiplied by itself or by some constant, then an appropri-

ate number of bits are extracted from the middle of the square in order to produce a hash address. If K bits are extracted, then 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 type of bit extraction scheme is used. The idea here is to use the middle bits of the square, which might be affected by all of the characters, or the whole bytes in the key in producing a hash address.

### 3.3.10 The Multiplicative Hash Method

A real number C between 0 and 1 is chosen in the multiplicative hash method <KNUT1, TENE1>. The hash function is defined as truncate(m * fraction(c * key)), where fraction(x) is the fractional part of the real number x (i.e., fraction(x) = x - truncate(x)). In other words, the key is multiplied by a real number (c) between 0 and 1. The fractional part of the product is used to provide a random number between 0 and 1 dependent on every bit of the key, and is multiplied by m to give an index between 0 and m-1. If the word size of the computer is 32 (2**5) bits, c should be selected so that 2**(2**5)*c is an integer relatively prime to 2**(2**5); c should not be too close to either 0 or 1. Also if r is the number of possible character codes, one should avoid values c such that fraction((r**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 0.6180339887, which equals (sqrt(5)-1)/2, or 0.3819660113, which equals 1-(sqrt(5)-1)/2.

### 3.3.11 The Radix Hash Method

In the radix hash method <MAUR1, LUM2>, a number representing the key is considered as a number in a selected base, e.g., base 11 rather than its real base. In the radix hash method, the resulting number is converted to base 10 for a decimal address. For example, the key 7,286 in base 10 is considered as 7,286 in base 11; therefore, 7,286 in base 11 becomes 9,653 in base 10, as is shown in the equation below:

7 * 11**3 + 2 * 11**2 + 8 * 11**1 + 6 = 9653 (in base 10)

Furthermore, the resulting number 9,653 can be divided by the number of buckets in the hash table. The remainder is then used as a hash address just like in the division method. This combination of two methods, the radix transformation and division methods, is derived from Lin's work <LUM1>. On the other hand, the number 9,653 may be multiplied by some fraction, then the fractional part of the product will be truncated in order to produce a number within the range of the hash table as a hash address. As

this method indicates, converting a number in one base to another base, hashes the bits. The hashed value is then used to produce a hash address.

### 3.3.12 The Random Hash Method

This random hash method <MAUR1> requires a statistically approved pseudo-random number generating function. After the key is encoded, the encoded word W is sent to the random number generating function as the seed. Then the random hash method applies division, or some other method, to the generated random number to produce a hash address. The distribution performance of this hash function is thus dependent on the chosen pseudo-random number generating function.

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

Recently, Pearson introduced a new hash method <PEAR1> for personal computers which lacks hardware multiplication and division functions. The major operations used in this hash method are exclusive-OR and indexed memory read and write. As shown in Figure 3-5, an auxiliary table(T) is used to contain 256 integers ranging from 0 to 255. Pearson's hash function receives a string of characters in ASCII code. Each character (C(i)) is represented by one byte that is used as an index in the range 0-255.

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

Figure 3-5. Pearson's Auxiliary Table T

As shown in Figure 3-6, each character of a key is exclusive-ORed with an indexed memory read (H(i-1)) in table H. The resulting byte is used to index the table T, and the indexed value in T is then stored to H(i) for the next iteration step. After the looping process is finished, the last indexed value (H(n)) from the table T becomes the hash address for the buckets ranging 0 through 255.

```
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 3-6. Pearson's Hash Algorithm

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

## 3.4 An Analysis of Distribution, Speed, and Cost

Table 3-1 shows each hash function's performance in terms of distributions on the three different data sets, in terms of speed when implemented either in software (SW) or in hardware (HW), and in terms of the cost of the hardware implementation of the hash function. For measurement of distribution, mean square deviation (MSD) is provided whenever a hash function is applied to the three different data

| Hash Method (D=Divisor) | <Distribution> MSD When Applied to RCN | GCN | RNS | <Speed> clocks SW | HW | <Cost> gates |
|---|---|---|---|---|---|---|
| Maurer's Shift-fold-loading | 3.95 Avr. | 4.06 Avr. | 3.81 Avr. | 420 | 70 | 384 |
| Berkovich's Hu-Tucker Code | 4.09 | 3.97 | 3.70 | 826(1) | 128(1) | 399 |
| Mapping | 3.93 Avr. | 4.06 Avr. | 4.07 Avr. | 96 | 3 | 120(2) |
| FS(0,10,20,30)<br>FS(0,11,22,25) | 4.20<br>4.03 | 3.96<br>4.46 | 4.27<br>4.88 | 44 | 2 | 192 |
| Algebraic Coding GF(2) | 4.41 | 4.62 | 3.77 | 452 | 48 | 390 |
| Pearson's Table Indexing | 20.63 | 21.23 | 21.15 | 82 | 82 | 280 |
| Digit Analysis (2 & 4 bytes) | 4.32<br>3.80 | 4.07<br>4.70 | 3.84<br>19.74 | 40(3) | 2(3) | 112<br>96 |
| Division (D=241) | 5.51 | 5.35 | 4.48 | 70 | 46<br>16(4) | 390<br>3360(4) |
| Folding (Fold-boundary) | 4.09 | 3.89 | 53.02 | 56 | 2 | 117 |
| Midsquare | 4.25 | 4.84 | 88.91 | 72 | 30<br>8(5) | 572<br>2796(5) |
| Multiplicative | 4.42 | 3.29 | 12.49 | 407 | 64<br>17(5) | 422<br>2892(5) |
| Radix | 3.97 | 4.05 | 12.36 | 650 | 390<br>285(5)<br>120(6) | 550<br>3234(5)<br>6498(6) |
| Random | 4.25 | 3.63 | 9.79 | 162 | 80<br>57(5)<br>26(6) | 470<br>3138(5)<br>6402(6) |

(1) Changeable due to variable length encoded key string.
(2) Sixteen 64x16 bits ROMs are also required.
(3) Analysis for digits is required beforehand.
(4) Faster but expensive since division array is used.
(5) Faster but expensive due to Wallace Tree for multiplication.
(6) Both Wallace Tree and division array are used.

Table 3-1. Performances of Hash Functions

sets: randomly chosen names (RCN), generally chosen names (GCN), and randomly chosen numeric strings (RNS). The number of clock cycles (clocks) is used in the measurement of the speeds of the hash coders. The cost of building a hardware hash coder is represented according to the number of gates needed. Finally, the performance of each hash function is discussed based on the experimental results.

### 3.4.1 Performance of Maurer's Shift-fold-loading method

The distribution performance of Maurer's shift-fold-loading hash method is measured in 10 different groups of eight bits among 128 resulting bits. The first group of eight bits for a hash address is made up of the 10th through the 17th resulting bits from the hash method. The second group is composed of the 20th through the 27th resulting bits from the hash method. The other groups are composed from bits 30 through 37 (30-37), 40-47, 50-57, 60-67, 70-77, 80-87, 90-97, and 100-107 resulting bits for the hash address. The mean square deviations of these groups are computed by requiring that the hash method be applied to the three data sets, such as RCN, GCN, and RNS, as shown in Table 3-2. The measured mean square deviations mostly are 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 hash

addresses with a good distribution.

| The MSDs of the Shift-fold-loading Hash Method | | | | | | |
|---|---|---|---|---|---|---|
| | Selected Bits for a Hash Address | | | | | |
| Data Set | 10-17 | 20-27 | 30-37 | 40-47 | 50-57 | 60-70 |
| RCN | 4.29 | 3.27 | 3.79 | 4.13 | 3.66 | 3.74 |
| GCN | 4.13 | 3.84 | 4.66 | 4.11 | 3.71 | 4.41 |
| RNS | 3.88 | 4.04 | 4.45 | 4.07 | 3.81 | 3.95 |

| Data Set | Selected Bits for a Hash Address 70-77 | 80-87 | 90-97 | 100-107 | Average of the 10 MSDs |
|---|---|---|---|---|---|
| RCN | 3.83 | 4.13 | 4.20 | 3.77 | 3.95 |
| GCN | 4.54 | 3.91 | 3.83 | 3.60 | 4.12 |
| RNS | 4.05 | 4.40 | 3.74 | 4.22 | 3.97 |

Table 3-2. Distribution Performance of Maurer's Shift-fold-loading Hash Method

The speed of this hash method is 420 clock cycles when implemented in software, and 70 clock cycles when implemented in hardware. Because the key registers that hold the 16-byte key cannot be assumed to be connected in a conventional processor, the software implemented shift-fold-loading hash coder takes considerable time in order to rotate some bits in a word register, and to copy some of the bits to the next word register. This hash method is also a hard-

ware oriented hash method which is similar to the mapping hash method.

When this method is implemented in hardware, a simple shift register, with adequate length to hold a four-word key, is provided for a fast shift operation. The hash method in hardware uses fast operations, such as shift and exclusive-OR. The decrease in speed is caused primarily by the log n times--n is the number of bits in a key (e.g., log 128 = 7)--of loading operations. Adding the repetitive loading operations in the hash algorithm may help to better the distribution of keys, but it also reduces the hash coder's performance in terms of speed.

The cost of this hardware hash coder is counted as 384 gates, which includes flip-flops in the shift register and exclusive-OR gates.

### 3.4.2 Performance of Berkovich's Hu-Tucker Code Hash Method

As shown in Table 3-1, 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 one of the best methods. Seemingly, this hash method is not data dependent since there is no distinguishable difference in the three mean square deviations. This data independency may result from the variable length of the Hu-Tucker code for each character. While partitions in a Hu-Tucker coded key are

folded, another randomization is added into the Hu-Tucker code conversion which already randomizes the value of the key, to 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 '111111001111110011111100'; whereas, that of the key 'EEE' is '010010010.' Hence, the key 'XXX' requires more folding steps, resulting in more time in the hash address calculation than the key 'EEE' does. Therefore, the speeds, 826 and 128 clocks, which are shown in Table 3-1, can be increased or decreased depending on every input key. Those clock cycle figures are obtained by using some medium size Hu-Tucker coded bit string. The major drawback in the speed performance of this hash method results from adding the number of bits in a Hu-Tucker code repeatedly to get the cumulative number of bits in the encoded bit string.

The hardware hash coder requires 399 gates owing to the assumption that eight bits are the maximum number of bits in Hu-Tucker code words. The register which contains the encoded bit string may have 128 flip-flops (8 bits in each of 16 characters). Moreover, an eight-bit counter register and several levels of exclusive-OR gates are helpful in increasing the speed of the address calculation.

### 3.4.3 Performance of the Mapping Hash Method

Distribution performances of the mapping hash method have been developed in cases when each ROM contains prime numbers and when each ROM contains random numbers. Tables 3-3A and 3-3B show the distribution performances in terms of mean square deviation (MSD) when the mapping hash method is applied to the three different data sets.

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

Table 3-3A. Distribution Performance of the Mapping Hash Method with Prime Numbers

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

Table 3-3B. Distribution Performance of the Mapping Hash Method with Random Numbers

As shown in the Tables 3-3A and 3-3B, mean square deviations hover around four, as do those of other relatively good hash methods. Since there is no distinguishable difference between using prime numbers and random numbers for each ROM, there is no clear reason to insist on solely prime numbers.

The results do not provide any clue regarding data dependency, since the mapping hash function distributes numeric string keys as well as other keys. Different groups of eight bits, e.g., 1-8, 2-9, 3-10, 4-11, 5-12, 6-13 bits, are extracted to compose a hash address (The 1-8 means bits 1 through 8 are selected.); there is no noticeable difference between the distribution performances of the various groups.

The major objective in developing a new hash method is to obtain a fast hash address calculation using any necessary hardware. By virtue of byte-by-byte parallel processing, with separate ROM and four levels of exclusive-OR gates for each character, the mapping hash method can produce a hash address within three clock cycles. Two clock cycles of the MC68030 processor are required for the memory read to retrieve a random number from the corresponding ROM, as is specified in the Motorola's users manual <MOTO1>. One clock cycle is taken for the calculation process for hash address bits through the four levels of exclusive-OR gates. As mentioned previously, the maximum gate delay is nine nanose-

conds and the clock frequency is set to 20 MHz (50 nanoseconds per a clock pulse width); thus, the address bit signal can pass through the four gate levels (4*9 = 36 <= 50 nsec) within a clock cycle. When this hash coder is implemented in software, the speed is 96 clock cycles, which is more than 32 times slower than the hardware hash coder's 3 clock cycles.

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

The performance of the author's additive mapping hash method, which is described in section 3.3.3, also is thoroughly examined in this dissertation. The mean square deviations of the additive hash method are 4.40, 3.91, and 3.58 when it is applied to the RCN, GCN, and RNS respectively. The additive mapping hash method shows competitive distribution performances when it is tested. This result supports the claim that addition and exclusive-ORing produce the same effect in randomizing the bit values. The hardware implementation of the additive mapping hash method does not substantially speed up the hash address calculation time, since 64 clock cycles are still required. The speed of the soft-

ware implemented additive mapping hash method is the same as that of software implemented exclusive-OR mapping hash method--96 clock cycles. The speed for both methods is based on the table of MC68030 instructions' execution time.

The architecture of the additive mapping hash coder would be the same as that of the carry lookahead adder <CAVA1> that is faster than the ripple carry adder. The number of gates needed to develop 16 bits of a carry lookahead adder is 182. The sixteen 64*16 bits ROMs, used in the hardware mapping hash coder, are no longer necessary, since in serial processing sixteen sets of 64 prime numbers, as a two dimensional array, can be stored in the main memory instead of in the 16 ROMs.

### 3.4.4 Performance of the Algebraic Coding Hash Method

The Galois Field GF(2) has been chosen in order to analyze the performance of the algebraic coding hash method. The distribution performance of this method (MSDs of 4.41, 4.62, and 3.77), as shown in Table 3-1, has been judged acceptable, because it does not show any data dependency. Although this is a hardware oriented hash method, the speed performance of 48 clock cycles in a hardware implementation is not as fast as the speed performance of the 3 clock cycles of the mapping hash method. If this method is implemented in software, it is relatively slow--452 clock cycles.

The number of gates needed to implement this method in hardware totals approximately 390.

### 3.4.5 Performance of the Digit Analysis Hash Method

The distribution performance of the digit analysis hash method <MAUR1, LUM1> is measured by using two types of encoded keys: 2 bytes and 4 bytes. Before the keys are hashed, they are scanned to provide information about the distribution of values of a key in each digit. The resulting statistics are shown in Tables 3-4A and 3-4B. The most skewed distributions--digits with a large deviation--are deleted until the number of remaining digits equals the hash address length of 8 bits.

As these statistics indicate, if the hash address bits consist of the selected bits from the digit analysis, then the subsequent distribution performance must be as good or better than other digit selections.

After investigation, it was found that the selected group of bits for the hash address in this hash method is not always the best group of bits. As shown in Table 3-1, in the case of the four byte encoded key, the MSD of RNS is 19.74 which is a relatively poor distribution performance. However, the MSDs in the case of the two byte encoded key are 4.32, 4.07, and 3.84. These findings indicate that this hash method may be data dependent.

| Statistics for the Digit Analysis Method (I) | | | |
|---|---|---|---|
| | Number of 1's and (Deviation) | | |
| Bit No. | RCN | GCN | RNS |
| Bit 1 | 489 (529) | 483 (841) | 502 (100)# |
| Bit 2 | 533 (441)* | 502 (100)@ | 521 (81)# |
| Bit 3 | 502 (100)* | 487 (625) | 478 (1156)# |
| Bit 4 | 484 (784) | 515 (9)@ | 517 (25)# |
| Bit 5 | 548 (1296) | 513 (1)@ | 0 (262144) |
| Bit 6 | 585 (5329) | 505 (49)@ | 0 (262144) |
| Bit 7 | 519 (49)* | 500 (144)@ | 0 (262144) |
| Bit 8 | 0 (262144) | 0 (262144) | 0 (262144) |
| Bit 9 | 501 (121)* | 506 (36)@ | 516 (16)# |
| Bit 10 | 534 (484)* | 523 (121)@ | 529 (289)# |
| Bit 11 | 498 (196)* | 526 (196) | 518 (36)# |
| Bit 12 | 524 (144)* | 542 (900) | 492 (400)# |
| Bit 13 | 500 (144)* | 504 (64)@ | 0 (262144) |
| Bit 14 | 420 (8464) | 494 (324) | 0 (262144) |
| Bit 15 | 486 (676) | 472 (1600) | 0 (262144) |
| Bit 16 | 0 (262144) | 0 (262144) | 0 (262144) |

Selected Bits in a Key (2 bytes) for Hash Address

RCN : 2, 3, 7, 9, 10, 11, 12, 13th bits (*)
GCN : 2, 4, 5, 6, 7, 9, 10, 13th bits (@)
RNS : 1, 2, 3, 4, 9, 10, 11, 12th bits (#)

Table 3-4A. Statistics for the Digit Analysis Method (I)

In the digit analysis hashing process, the keys in the input data file have to be scanned twice, once for digit analysis and once for hashing. The speed of this hash method is 40 clock cycles for software implementation, and 2 clock cycles for hardware implementation. However, this method only works for a static input file. When a static input file is read and digits are selected by the analysis, the

| Statistics for the Digit Analysis Method (II) | | | |
|---|---|---|---|
| | Number of 1's and (Deviation) | | |
| | RCN | GCN | RNS |
| Bit 1 | 536 (576) | 492 (400) | 0 (262144) |
| Bit 2 | 508 (16)* | 520 (64) | 521 (0)# |
| Bit 3 | 498 (196) | 518 (36) | 509 (9)# |
| Bit 4 | 500 (144) | 517 (25)@ | 474 (1444) |
| Bit 5 | 473 (1521) | 475 (1369) | 0 (262144) |
| Bit 6 | 689 (31329) | 744 (53824) | 0 (262144) |
| Bit 7 | 435 (5929) | 429 (6889) | 0 (262144) |
| Bit 8 | 0 (262144) | 0 (262144) | 0 (262144) |
| Bit 9 | 514 (4)* | 522 (100) | 516 (0)# |
| Bit 10 | 467 (2025) | 517 (25)@ | 522 (100)# |
| Bit 11 | 516 (16)* | 510 (4)@ | 514 (4)# |
| Bit 12 | 481 (961) | 498 (196) | 436 (5766) |
| Bit 13 | 430 (6724) | 462 (2500) | 0 (262144) |
| Bit 14 | 329 (33489) | 275 (56169) | 0 (262144) |
| Bit 15 | 483 (841) | 443 (4761) | 0 (262144) |
| Bit 16 | 0 (262144) | 0 (262144) | 0 (262144) |
| Bit 17 | 519 (49)* | 509 (9)@ | 502 (100)# |
| Bit 18 | 505 (49)* | 512 (0)@ | 509 (9)# |
| Bit 19 | 512 (0)* | 535 (529) | 495 (289) |
| Bit 20 | 520 (64)* | 516 (16)@ | 443 (4761) |
| Bit 21 | 483 (841) | 424 (7744) | 0 (262144) |
| Bit 22 | 366 (21316) | 399 (12769) | 0 (262144) |
| Bit 23 | 502 (100) | 515 (9)@ | 0 (262144) |
| Bit 24 | 0 (262144) | 0 (262144) | 0 (262144) |
| Bit 25 | 501 (121) | 502 (100) | 500 (144) |
| Bit 26 | 509 (9)* | 518 (36)@ | 505 (49)# |
| Bit 27 | 522 (100) | 522 (100) | 526 (196) |
| Bit 28 | 493 (361) | 496 (256) | 420 (8464) |
| Bit 29 | 418 (8836) | 462 (2500) | 0 (262144) |
| Bit 30 | 353 (25281) | 335 (31329) | 0 (262144) |
| Bit 31 | 501 (121) | 545 (1089) | 0 (262144) |
| Bit 32 | 0 (262144) | 0 (262144) | 0 (262144) |

Selected Bits in a Key (4 bytes) for Hash Address

RCN : 2, 9, 11, 17, 18, 19, 20, 26th bits (*)
GCN : 4, 10, 11, 17, 18, 20, 23, 26th bits (@)
RNS : 2, 3, 9, 10, 11, 17, 18, 26th bits (#)

Table 3-4B. Statistics for the Digit Analysis Method (II)

keys are hashed very fast. The cost of building this hash coder in hardware is the least expensive, either 112 or 96 gates for encoding, since it does not include any hardware function for mathematical operation.

### 3.4.6 Performance of the Division Hash Method

The distribution performance of the division hash method <BUCH1, MAUR1, LUM1> varies, depending on the chosen divisor which is close to the number of buckets, as is shown in Table 3-5.

If an inappropriate divisor is chosen, a data dependency problem may occur. In this experiment, the divisors which are greater than the number of buckets in a table (i.e., 256) are also tested. The divisor 257 is a nonprime number with prime factors less than 20, as recommended by Lum and his colleagues, but it shows very poor distributions (MSDs of 5.67, 11.95, and 122.99). As Maurer and Buchholz suggested <MAUR2, BUCH1>, using the largest prime number, (i.e., 241) that also is smaller than the number of buckets, as the divisor, yields better results (MSDs of 5.51, 5.35, 4.48).

The speed of the division method is relatively fast for both the software implementation (70 clock cycles) and the hardware implementation (46 or 16 clock cycles). The two major means of implementing the division hash method in

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

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

Table 3-5. Distribution Performance of the Division Hash Method

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

3360 gates.

The ordinal numbers of each character in a key are added and the sum is divided by the number of buckets in order to calculate the remainder; this process is referred to as the additive division method. This method also has been analyzed (MSDs are 3.97, 3.91, and 86.76). When the additive division is applied to the data set RNS, a poor distribution (MSD 86.76) results because the cumulative numbers (or SUMs) from the numeric character strings are not large enough when compared to the number of buckets. Therefore, the additive division method can only be used when the average ratio of sums and the number of buckets (sums/number of buckets) is sufficiently large.

Several other researchers <BUCH1, LUM1, RAMA1> conducted experiments on typical key sets in order to discover the ideal hash method. Their overall conclusions verify that the simple method of division seems to be the best key to address transformation technique when computational time is not critical. Nevertheless, in this survey of hash methods, the division method is not highly recommended, since either the mapping or the additive mapping method can be used instead, depending on the application environment. In the application, where fast hash address calculation is not required, the author's additive mapping method is superior to the division method. When using the additive mapping method, one need not worry about selecting a correct divi-

sor; one need only divide the sum or combination by the number of buckets in order to arrive at a remainder for a hash address. On the other hand, when the speed in address calculation is imperative and the number of buckets can be 2**n, then a hardware hash coder is needed, and the mapping hash coder which is faster and cheaper than the division hash coder is thus recommended.

### 3.4.7 Performance of the Folding Hash Method

There are several different types of folding hash methods. In this experimental environment, the fold-boundary method, as described by Maurer <MAUR1>, is simulated in order to show the distribution performance. The MSDs are 4.09, 3.89, and 53.02 when the bits (11 through 18, which are exclusive-ORed twice) are extracted to compose a hash address.

As shown in Table 3-6, the distribution performance on the randomly chosen numeric strings (RNS) data set is poor. The bits, in particular, the 11th through the 18th, the 12th through the 19th, and the 13th through the 20th, are used to represent a hash address; however, data dependency is still reflected in the distribution. A deliberately designed folding method--rotating bits in a different way and then folding them together--might prevent the data dependency problem in the key distribution. In this dissertation, an effort is

made to introduce a reasonable fold-shifting hash technique for better key distribution, such as FS(0,10,20,30), as was mentioned in section 3.4.8.

| The MSDs of the Folding Method | | | |
|---|---|---|---|
| Bits Selected | RCN | GCN | RNS |
| 18 ... 11 | 4.09 | 3.89 | 53.02 |
| 19 ... 12 | 3.72 | 3.89 | 53.86 |
| 20 ... 13 | 3.63 | 3.62 | 56.23 |

Table 3-6. Distribution Performance of the Fold-boundary Hash Method

### 3.4.8 Performance of the Fold-shifting Hash Method

The distribution performances of the author's fold-shifting hash method, in particular, FS(0,10,20,30) and FS(0,11,22,25), are as good as those of other acceptable hash methods. But other selected fold-shifting methods, such as FS(0,12,17,29), FS(0,13,18,31), and FS(0,15,22,29), show a data dependency problem, such that the distribution performance on the RNS data set is not compatible with the distribution performance on the RCN and GCN data sets, as is demonstrated in Table 3-7. Therefore, careful selection of the number of partitions and the number of rotated bits is required.

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

Table 3-7. Distribution Performances of Various Fold-shifting Hash Methods

Therefore, the FS(0,10,20,30) is recommended for a hardware implemented hash coder, since the hash operation takes only two clock cycles and the number of gates required in a hash coder is 192, which is relatively inexpensive.

### 3.4.9 Performance of the Midsquare Hash Method

As shown in Table 3-1 and Table 3-8, the mean square deviations (4.25, 4.84, and 88.91) of the midsquare method <MAUR1, LUM2> reveals that this method has a data dependency problem; that is, in cases where the keys are similar in some form, this method has a high potential for producing more key clusterings.

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

Table 3-8. Distribution Performance of the Midsquare Hash Method

When this hash method is implemented in software, the hashing operation requires 72 clock cycles. The speed of a hardware midsquare hash coder which uses a sequential add/

shift multiplier <CAVA1> is 30 clock cycles. The speed of the coder can be improved to eight clock cycles if a Wallace's fast multiplier is used <WALL1>. However, the cost of the hardware hash coder increases from 572 to 2796 gates for the speed gain.

### 3.4.10 Performance of the Multiplicative Hash Method

According to the MSDs (4.42, 3.29, 12.49) of the multiplicative hash method <KNUT1, TENE1> in Table 3-1, the distribution performance of the multiplicative hash method is open to doubt. This method may have a data dependency problem--since, as is indicated in the table, the MSD of 12.49 on the numeric strings (RNS) data set is one of poorest results for a hash method.

The software implemented multiplicative hash coder is relatively slow (407 clock cycles); however, if Wallace's fast hardware multiplier is built into the hash coder, it can improve the speed to 17 clock cycles with the cost of 2,892 gates. Thus, the sequential add/shift multiplier may 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 422 gates.

### 3.4.11 Performance of the Radix Hash Method

The distribution performance of the radix hash method

<MAUR1, LUM2> is similar to that of the multiplicative hash method. The distribution performances (MSDs 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). However, this hash method also has demonstrated a potential to perform poorly when keys in a data set are similar.

The radix hash method is one of the most time consuming methods since it takes extra time to convert 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), and it increases the cost (550, 3234, and 6498 gates, respectively), as has been disclosed in Table 3-1.

### 3.4.12 Performance of the Random Hash Method

The distribution performance of the random hash method <MAUR1> 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 among the lowest of those tested.

The speed performance, implemented in software, is measured at 162 clock cycles, which includes the computation time of a multiplication, a division, and an addition. The hardware hash coder for the random method may be implemented in many different ways. An economic hardware hash coder, which is not equipped with a division array or Wallace's multiplier, takes about 80 clock cycles in a hash address calculation and requires 470 gates to provide a sequential multiplier and divider, carry lookahead adder, and exclusive-OR gates for encoding.

If the hash coder uses a Wallace fast multiplier, it can speed up the address computation to 57 clock cycles; however, the cost rises to 3,138 gates. If the division array is added for fast division, the number of clock cycles in the address calculation drops to 26; however, the expense is subsequently increased to 6,402 gates.

### 3.4.13 Performance of the Pearson's Table Indexing Hash Method

Pearson's table indexing hash method appears to be erratic owing to its poor distribution performance (MSDs: 20.63, 21.23, and 21.15). Hardware implementation of this hash function would not improve the speed of address calculation. The measured speed of address calculation is 82 clock cycles in both hardware and software implementations.

# CHAPTER 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 of the HIMOD database computer. In the next section, the connection between the host processor and the join database coprocessor is described; each of their functionalities in performing the join operation also is illustrated. The architecture of the host processor and the software back-end are described in section 4.3. The architecture of the hardware back-end is described in block diagrams in the final section.

## 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 not only has reliable, data independent, and relatively good key distribution, but it also takes only three clock cycles to transform a key to an address if the mapping hash coder is implemented in hardware. The performances of the mapping hash method, in terms of both the key distribution and the speed of the mapping hash coder, are highly persuasive.

In this application environment, the hash coder is the major component in the database coprocessor which is used as a filter device. When long queries are being executed on a large database, a series of millions and millions of data are waiting to be hashed out through that filter. The speed in a hash address calculation should be a crucial factor in selecting a hash method.

The mapping hash technique is designed not only for good distribution but also for fast address calculation. The parallel processing transforms each character into a number and calculates each bit value in a hash address by means of hardware, in order to produce a hash address within three clock cycles. Other hash methods cannot take advantage of such effective parallel processing because of the algorithmic nature of their hash address calculation. For example, some of the well-known hash functions, such as the midsquare and the fold-boundary, show data dependency problems. Other hash functions, like the multiplicative, the radix, and the random, show signs that they may perform poorly for specific data sets.

The mapping hash coder in hardware is relatively inexpensive compared to other hardware hash coders. It requires 120 gates and sixteen 64*16 bits ROMs, since the mapping hash function does not use the complex mathematical operations, like multiplication and division, that other hash methods, such as midsquare, multiplicative, radix, random,

and algebraic coding, do.

Another requirement of a hash function for this application specifies that the database coprocessor must have five statistically independent hash coders; five hash addresses are produced within the same period of time and must be totally independent of each other. The reason for this requirement will be explained in the next section in which the filtering technique which employs five functionally different hash coders will be described. For this particular requirement, the mapping hash method is advantageous, since based on the stored contents (selected prime numbers) of the ROMs, each hash coder calculates a hash address in its unique way. Because each of the five hash coders has a different set of numbers, the five hash addresses generated at a time are independent of each other. The address calculation time for each hash coder is always the same. This characteristic of statistical independence becomes 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 brute-force; therefore, claiming that each hash function is 100 percent functionally different is difficult. In addition, it also is hard to maintain the same address calculation time for each modified hash coder.

An alternative hash method might be folding of the four

rotated keyword, e.g., FS(0,10,20,30). It is extremely fast and inexpensive when this method is implemented in hardware. The distribution performance of this method seems good, but it is not very reliable in terms of data dependency. Nonetheless, there has been an effort to discover five statistically independent hash functions using this method. However, data independence remains an uncertainty, since some of them have shown a data dependency problem.

Consequently, it is concluded that the mapping hash coder in hardware satisfies the three requirements of distribution, speed, and cost; moreover, it provides the property of statistical independence in each hash function. As a result, this method has been adapted for the database hash coder in the database computer HIMOD.

## 4.2 An Overview of HIMOD Architecture

As mentioned previously, the primary goal of this dissertation research 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 of this dissertation research is to maximize the filtering effect in the join process, in order that the database coprocessor may be used as a filter device. In this way, the unnecessary data are filtered by the filter as soon as they are detected and only

the tuples included in the resulting relation will be transmitted to the host processor. The transmitted source tuple(s) and target tuple(s) are then merged by the host processor. Therefore, parallelism is exploited in the join operation so that the filtering process and the merging process are concurrently executed by the back-end and the host respectively, as is indicated in Figure 4-1.

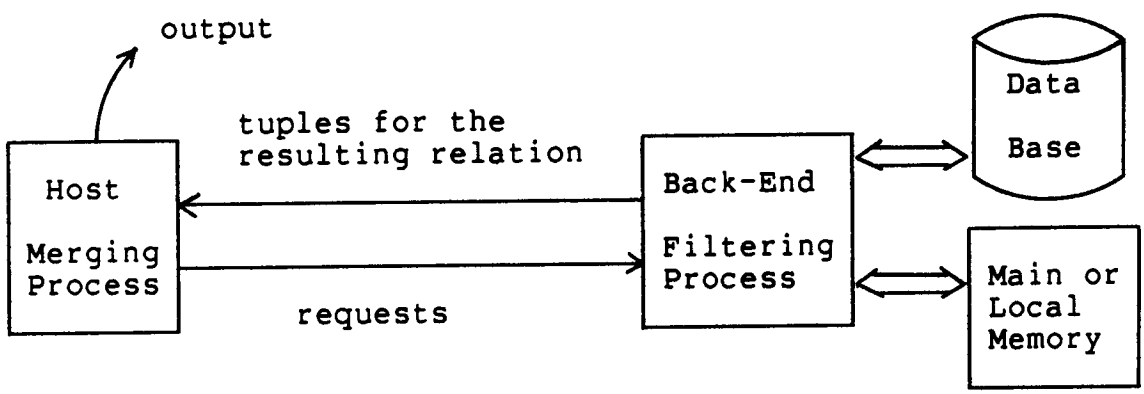


Figure 4-1. Execution of the Relational Join in HIMOD

The main idea behind the new relational join in HIMOD contends that the tuples, which the host processor receives, are the only ones necessary in producing a resulting relation. There are only a small number of tuples in most cases; therefore, the host processor is not burdened with carrying many join attributes and comparing them for a match. The filtering scheme in HIMOD is accomplished by the stack oriented filter technique (SOFT) and the new hash-based join

algorithm which will be more 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, which is defined as the M68000 coprocessor interface <MOTO1>. A back-end processor adds additional database instructions, plus additional registers and data types to the programming model not directly supported by the host processor architecture. The necessary interactions between the host processor and the database coprocessor become 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 to be separate from the host processor.

The connection between the host processor unit (HPU) and the database coprocessor (DBCP) develops from a simple extension of the M68000 bus interface. The DBCP is connected as a coprocessor to the host processor, and a chip select signal, decoded from the host processor function codes and address bus, selects the DBCP. The host processor and the coprocessor configuration is shown in Figure 4-2.

All communications between the HPU and the DBCP are performed with standard M68000 family bus transfers <MOTO1>.

The DBCP is designed to operate on a 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.

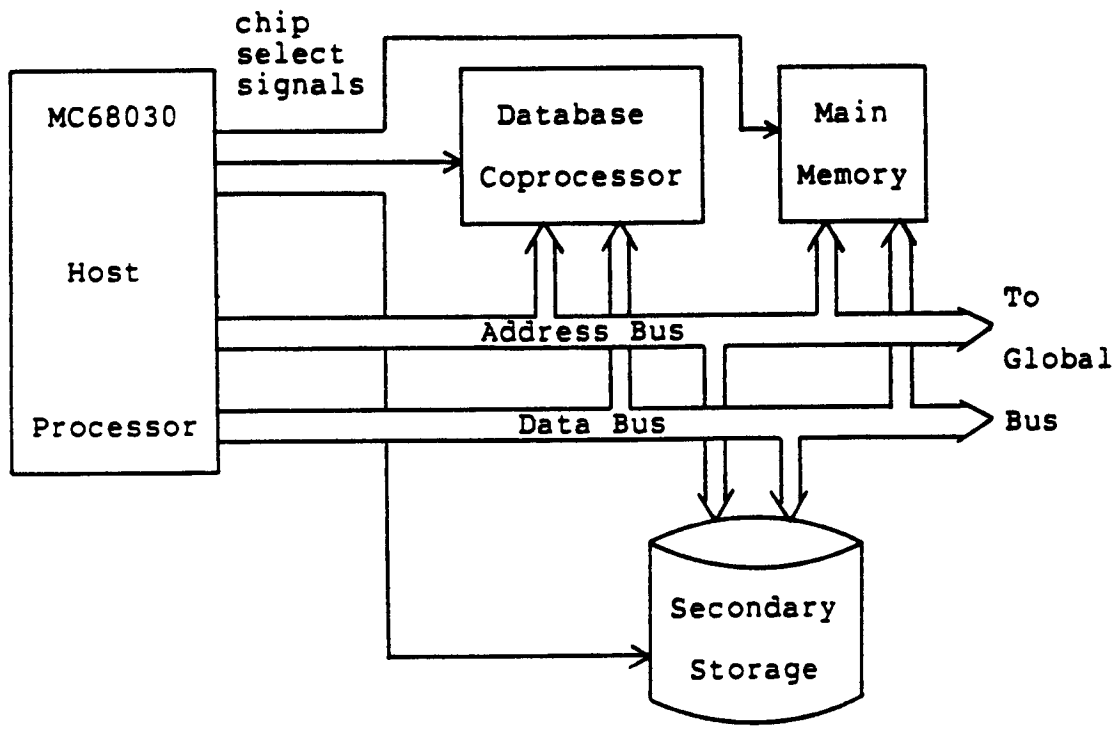


Figure 4-2. Coprocessor Configuration

## 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 <MOTO1> are briefly illustrated here. The MC68030 is a second-generation full 32-bit enhanced microprocessor from

Motorola. The MC68030 combines a central processing unit, 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. It is implemented with 32-bit registers and data paths, 32-bit addresses, a rich instruction set, and versatile addressing modes.

The MC68030 enhanced microprocessor provides the non-multiplexed bus structure with 32 bits of address and 32 bits of data. The MC68030 bus has a controller that supports both asynchronous and synchronous bus cycles, as well as, bus 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.

In the MC68030 enhanced microprocessor, 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.

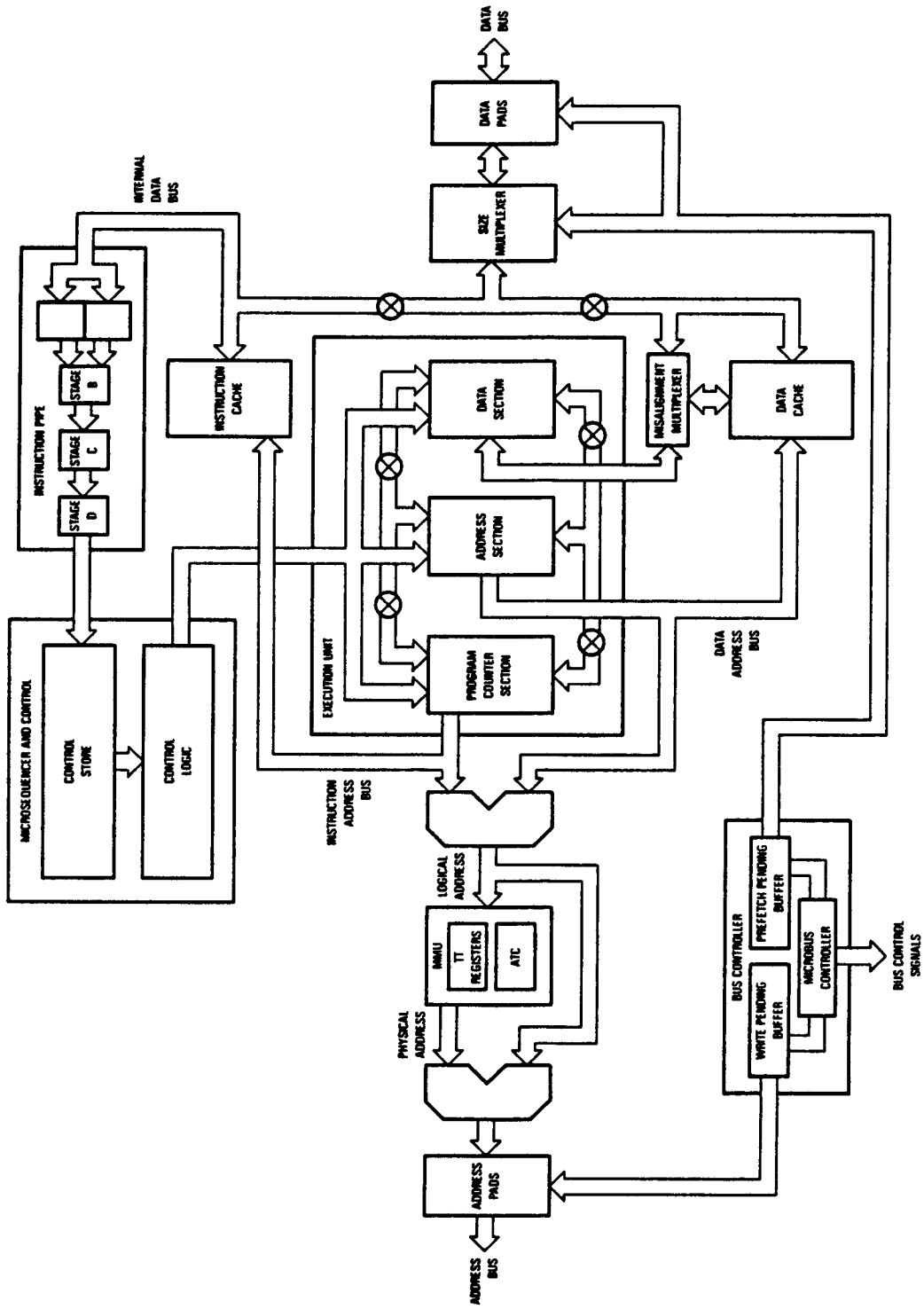


Figure 4-3. Motorola MC68030 Block Diagram <MOTO1>

As described in the MC68030 user's manual <MOTO1>, the major 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. Two 32-bit supervisor stack pointers and ten special purpose control registers,
5. 256-byte instruction cache and 256-byte data cache that can be accessed simultaneously,
6. Paged memory management unit that translates addresses in parallel with instruction execution and internal cache accesses,
7. Two transparent segments which allow untranslated access to physical memory to be defined for systems that transfer large blocks of data between predefined physical addresses, e.g., graphics applications,
8. Pipelined architecture with increased parallelism that allows accesses to internal caches to occur in parallel with bus transfers and instruction execution to be overlapped,
9. Enhanced bus controller that supports asynchronous bus cycles (three clocks minimum), synchronous bus cycles (two clocks minimum), and bus data transfers (one clock minimum) all to the physical address space,
10. Dynamic bus sizing supports 8-, 16-, 32-bit memories and peripherals,
11. Support for coprocessors with the M68000 coprocessor interface; e.g., full IEEE floating-point support provided by the MC68881/MC68882 floating-point coprocessors,
12. 4-gigabyte logical and physical addressing range,
13. Implementation of Motorola's HCMOS technology that allows CMOS and HMOS (High Density NMOS) gates to be combined for maximum speed, low power, and optimum

die size,

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 dedicates only database management functions; therefore, the contents of microsequencer and control store should be entirely replaced with the microcodes of database operations. Furthermore, there is no hardware modification or enhancement on the database coprocessor except for the interface unit; innovative software architecture is implemented on the back-end instead.

## 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 that produce five statistically independent hash addresses simultaneously within three clock cycles,

3. Two single-bit wide (256 bits) RAMs connected to each hash coder to keep the records of generated hash addresses,

4. Five hash address comparators which are attached to the corresponding hash coders, tell whether only one kind of key (or join attribute) has passed through the filter or not,

5. Condition code to check if only one kind of hash address has been produced.

As shown in Figure 4-4, the database coprocessor is internally divided into three processing elements: the bus interface unit, the coprocessor control unit, and the filter unit. The bus interface unit communicates with the host processor, and the coprocessor control unit sends control signals to the bit array filter unit in order to execute the intended database operation. For both the bus interface unit and the processor control unit, the DBCP uses the conventions of the MC68881 and MC68882 floating-point coprocessor chips <MOTO2>.

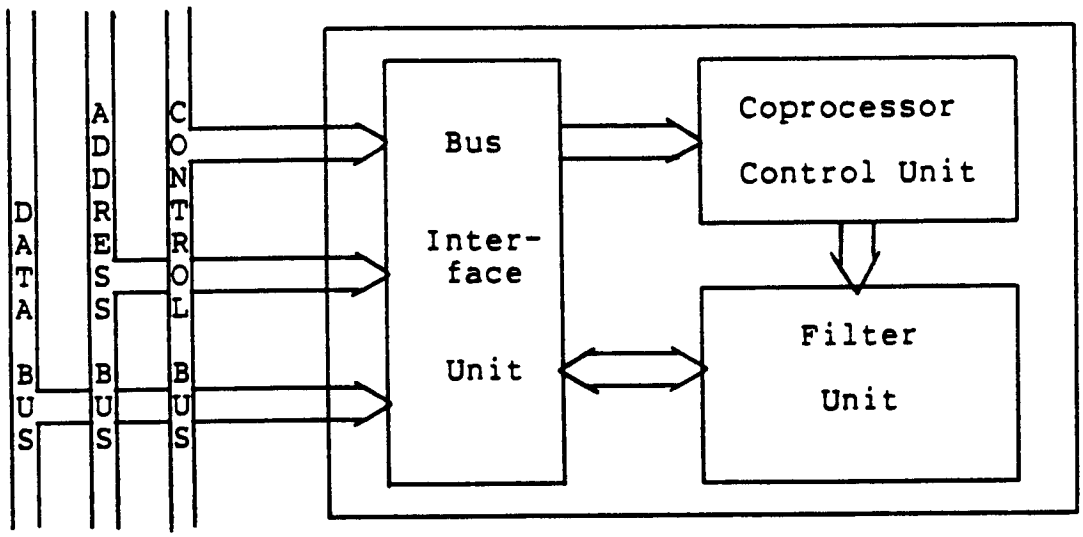


Figure 4-4. DBCP Simplified Block Diagram

### 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 that are used to monitor the status of communications with the host processor. The CIRs are addressed by the host processor in the same manner as memory. All communications between the host processor unit and the DBCP are performed with standard M68000 family bus transfers <MOTO1>. The M68000 family coprocessor interface is implemented as a protocol of bus cycles during which the host processor reads and writes to these CIRs. The MC68030 host processor implements this general purpose coprocessor interface protocol in both hardware

and microcode.

When the host processor detects a DBCP instruction, e.g., a join operation, the host processor writes the command word of the instruction to the memory-mapped command CIR and then reads the response CIR. In this response, the bus interface unit encodes requests for service required by the host processor in order 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; thus, the synchronization during host processor and DBCP communication can be accomplished. Concurrent or nonconcurrent instruction execution is determined based on the nature of each coprocessor instruction. While the execution of most DBCP instructions may overlap with the execution of host processor instructions, concurrency is completely transparent to the programmer. Therefore, from the programmer's view, the host process and the DBCP appear to be integrated onto a single chip.

It is worth noting that the DBCP does need not 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. Because the M68000 family coprocessor interface also permits coprocessors to be bus masters, the DBCP functions as one. That is, the DBCP

can fetch all tuples or attributes and store all results. In this manner, the DBCP as a filter device, effectively performs the intended database operation by filtering unnecessary data. These types of coprocessors are referred to as DMA coprocessors. The DBCP DMA coprocessor operates as a bus slave while communicating with the host processor across the coprocessor interface. The DBCP may also have the ability to operate as a bus master, thereby directly controlling the system bus. This type, based on bus interface capability, is called non-DMA coprocessors which 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; therefore, the DBCP needs to be implemented as a DMA coprocessor. In the end, the DBCP provides all the control, address, and data signals necessary to request and obtain the bus, and to then perform 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 either executes microinstructions or awaits completion of accesses that are necessary to continue executing microcode. The microsequencer sometimes controls the bus controller, which is responsible for all bus activ-

ity. The microsequencer also controls instruction execution and internal processor operations, such as setting condition codes and calculating effective addresses. The microsequencer provides the microinstruction decode logic, the instruction decode register, the instruction decode PLA, and it determines the "next microaddress" generation scheme for sequencing the microprograms.

The microcode ROM contains the microinstructions, which specify the steps through which the machine sequences and which control the parallel operation of the functionally equivalent slices of the filter unit.

### 4.4.3 Filter Unit

One of the main tasks of the DBCP is to release the host from tedious database manipulation for the relational join by filtering tuples that do not have any potential for inclusion in the resulting relation. To this end, the DBCP sends only the potential tuples to the host processor. The filter unit of the DBCP is the heart of the coprocessor in determining unnecessary data and discarding them. There are several approaches in implementing a filter device <BANC1, BABB1, HSIA1>; however, their fundamental concepts are the same. These filter devices trap irrelevant data when it is transferred from the secondary storage device to the main memory.

As shown in Figure 4-5, the filter unit of the DBCP includes an address filter, five functionally different mapping hash coders with associated bit array store (BAS) and associated hash address comparator (HAC), and an AND module (see Figure 4-6 for more details).

The bit array store (BAS) includes two single-bit wide random access memories (256 bits RAMs). One RAM (source RAM) is for tuples in a source relation, and another (target RAM) is for tuples in a target relation. Each bit in a RAM is addressed by a hash address. Each BAS is connected with a hash address register in an associated hash address comparator (HAC). The hash address register is equipped with an increment function so that the address register will keep track of the next bucket address to be processed, and will feed it to the connected BAS. Therefore, each bit array store has a built-in multiplexer to select the right address at any time as is shown in Figure 4-7. The controller sends signals to the control lines of the multiplexer for the right selection of an address. The controller also sends memory write signals to both source and target single-bit wide RAMs. Therefore, when the tuples in the source relation are scanned, the single-bit wide source RAM is marked based on the hash address from the hash coder.

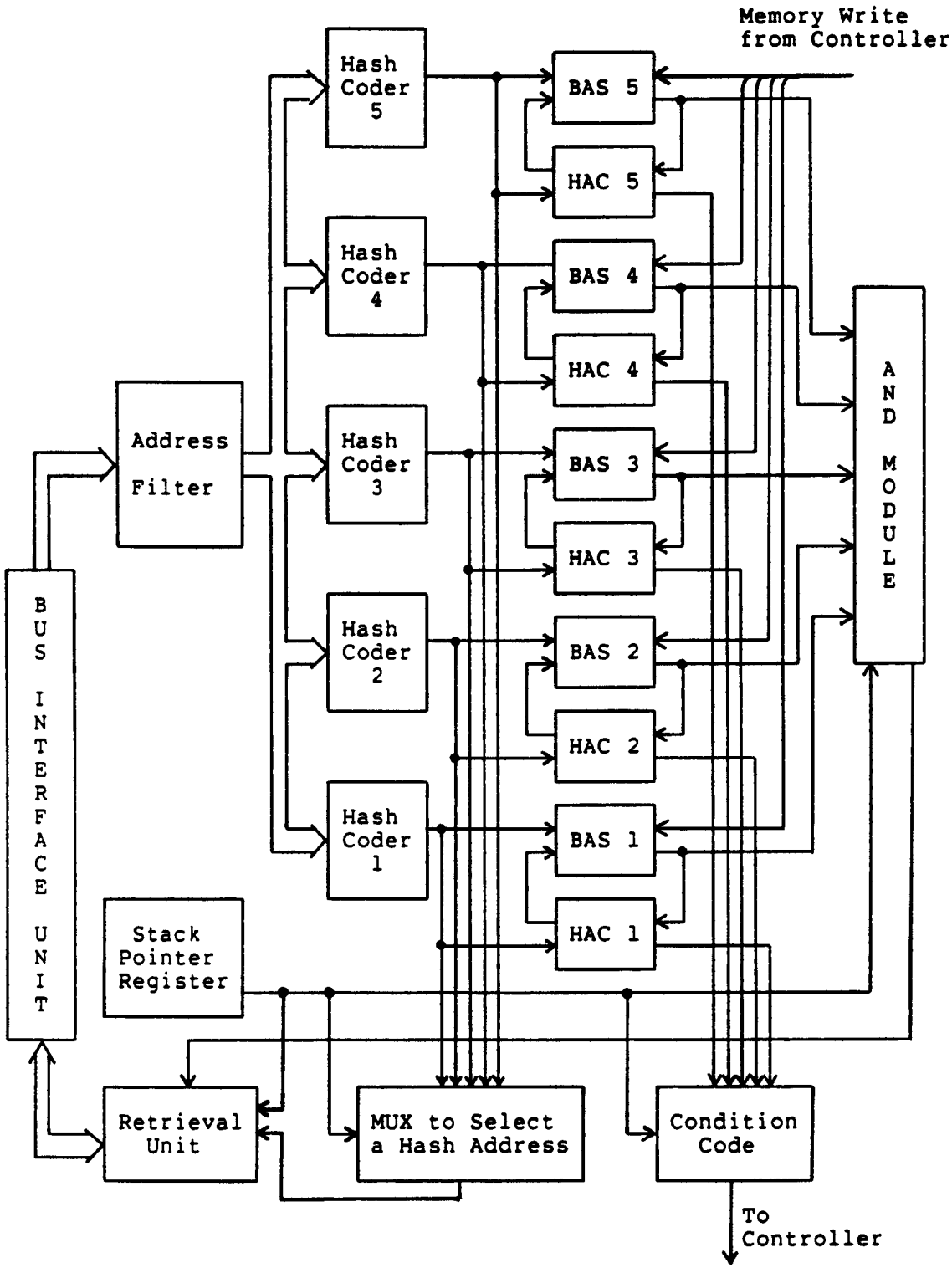


Figure 4-5. The DBCP Architecture (Filter Unit)

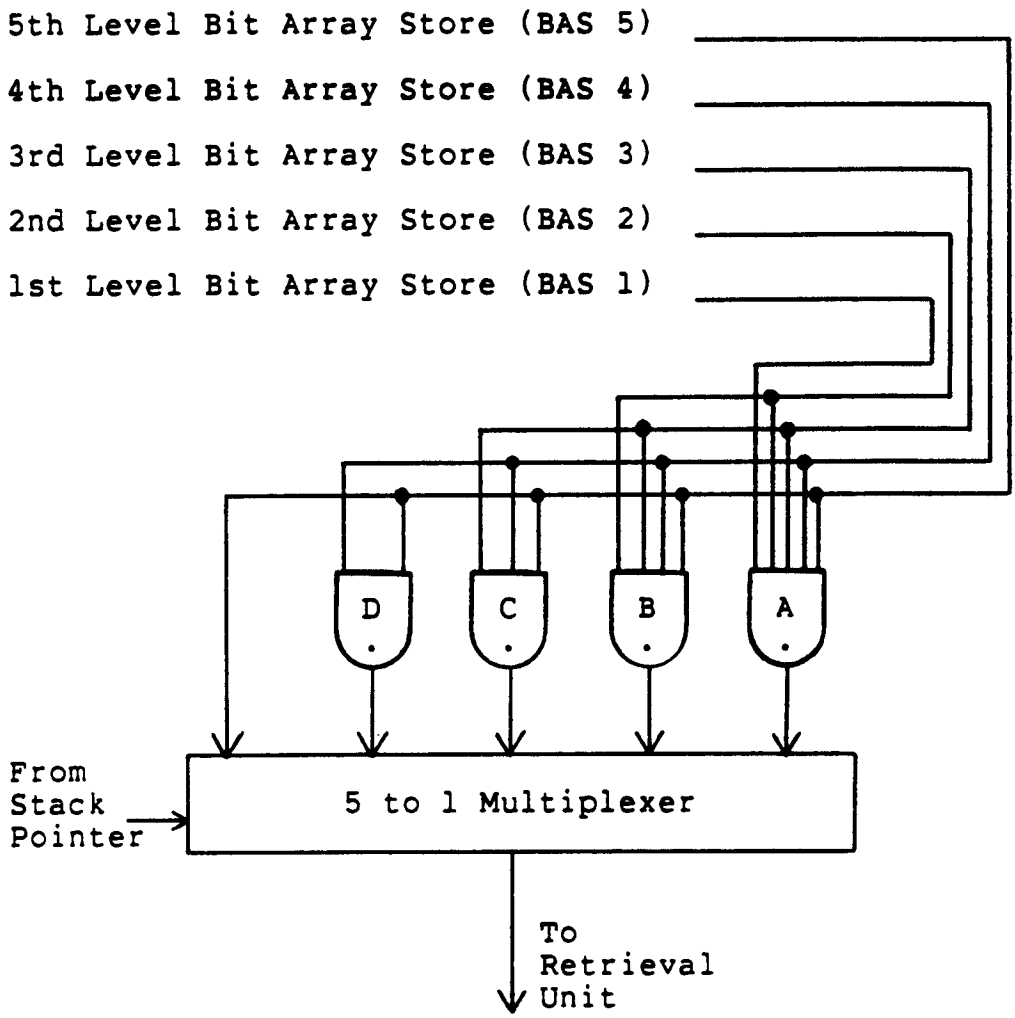


Figure 4-6. AND Module

By the same system, when the tuples in the target relation are scanned, the single-bit wide target RAM is marked instead. The single-bit output from the source RAM to the AND module is examined in order to filter unnecessary target tuples. When the multiplexer in the BAS selects a hash address from the corresponding HAC, the hash address is used

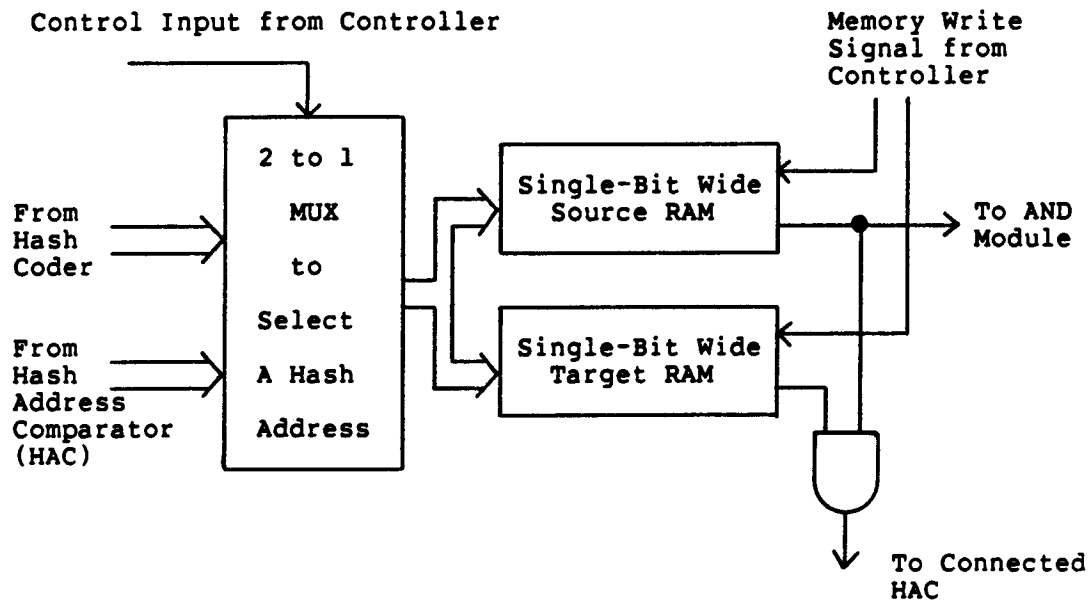


Figure 4-7. Bit Array Store (BAS)

to determine if one of source and target buckets is empty. This can be done by detecting hash-addressed bits if both the source RAM and target RAM are '1.' The single-bit outputs from the source RAM and the target RAM are then logically ANDed. The resulting single-bit output is sent to the corresponding HAC in order to determine whether or not the tuples in the source and target buckets have to be processed. If one of the buckets is empty, then tuples in those buckets will be eliminated.

The HIMOD database computer adapts the hashed address bit array stores filtering technique which was introduced in the Content Addressable File Store (CAFS) database machine <BABB1>. This filtering technique has demonstrated dramatic

improvement in all join algorithms without substantially increasing hardware cost <DEWI3, QADA2, SHAP1, VALD2, SCHN1>. Even though there are explicit details about the hashed address bit array stores filtering technique in Chapter 5, it is necessary, at this point, 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; in turn, each value of the join attribute is transformed into five hash addresses by five functionally different hash coders. To simplify the explanation, it is assumed here that the current stack level is at the bottom. These five addresses are used to mark the single-bit wide source RAMs in the corresponding BASs. Then the machine reads the target relation, and the five hash coders again hash each value of the join attribute. By using the five hash addresses, HIMOD verifies if the hash-addressed bits in the five source RAMs in the corresponding BASs have already been set. If all five bits have already 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, they are sent to the host computer to produce the tuples of the resulting relation. On the other hand, the unwanted tuples are detected and discarded by the DBCP, since they will not be included in the result-

ing 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)), M2(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 the corresponding single-bit wide source RAMs. The hash addresses set the corresponding bits of the source RAMs to '1.' After scanning keys and setting up bits in the source RAMs, the output of the five source RAMs are logically ANDed, when key K is read from the source RAMs in the BASs. If the output of the AND gate is '1,' then the key K is believed to have been hashed before. Otherwise, key K has not been hashed and it is possible to set the source RAMs in the BASs.

To write keyword K to the source RAMs in the BASs:

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

To read keyword K from the source RAMs in the BASs:

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

As assumed, the current stack level is at the bottom in this case. Thus, the 5 to 1 multiplexer controlled by the stack pointer register selects the output from the AND gate (A) in the AND module as shown in Figure 4-6. This output is sent to the retrieval unit that determines whether the tuple is unnecessary or not, based on the output from the AND module. If the tuple is unnecessary, it is discarded immediately.

The discussion above shows how the hashed address bit array stores filtering technique works in the HIMOD database computer. The hardware structure that enhances this filtering technique also should be explained. The architecture of the DBCP is characterized by a stack oriented structure of the five BASs. If there are any bit array stores lower than the current BAS, they are saved in the stack and deactivated during the filtering process. The current and higher-than-current BASs participate in the filtering process. The contents of participating BASs are cleared first and the bits in the BASs marked as the hash addresses are then produced. When a BAS is saved in the stack, a file or a list of input tuples are divided and distributed into the addressed buckets in the hash table according to the prior level hash coder in the stack. The divided list of source tuples and the list of target tuples are passed through the filter again using the current and higher BASs if their join condition attributes are not detected as identical. Thus the

source and target relations are divided repeatedly, discarding unwanted tuples, until the DBCP determines that the partitioned lists of the source and target tuples have the same join attribute. Ultimately the partitioned lists of the source and target tuples are sent to the host processor for final screening and then merge in order to produce the resulting tuples.

To efficiently determine whether or not the scanned source tuples and target tuples have the same join attribute, a hash address comparator (HAC) is attached to each of the five hash coders. The HAC is designed so that it sends a signal to the controller to stop dividing the tuples, as is explained below.

The HAC consists of an address register (or hash address register) which keeps a record of the first hash address produced by the corresponding hash coder and the number of exclusive-OR gates, the OR gate, and the JK flip-flop. Each incoming hash address is compared with the first produced hash address, as illustrated in Figure 4-8.

In order to load the first hash address, a controller sends a signal ('1') to load the first produced hash address into the hash address register. Once the first hash address is loaded, the controller does not allow other hash addresses to be loaded into the hash address register.

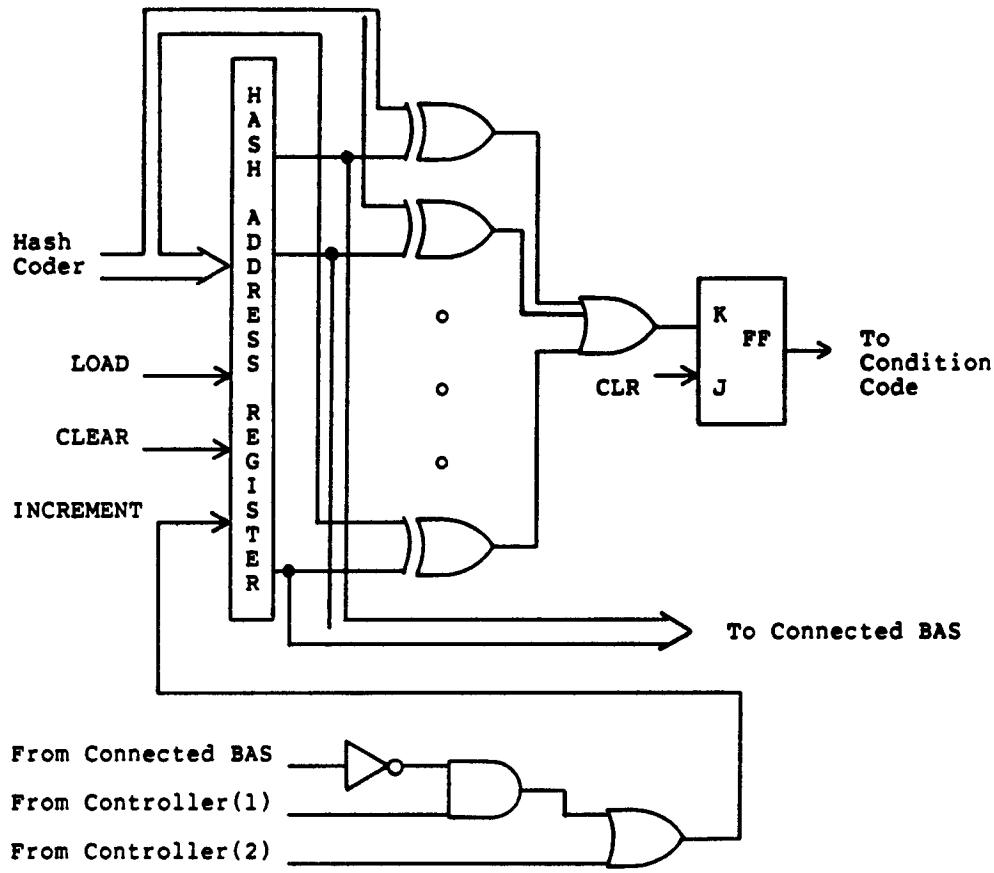


Figure 4-8. Hash Address Comparator (HAC)

In each HAC, the same number of exclusive-OR gates, as the number of bits in a hash address register, are needed. The first bit of the address loaded in the hash address register and that of an incoming hash address are inserted into the first exclusive-OR gate. If both are the same, the output of the exclusive-OR gate is '0.' If they are not the same; that is, if one input bit is '1' and another is '0,' then the output is '1,' and it is passed to the OR gate.

The OR gate simultaneously receives all the resulting output signals from those exclusive-OR gates. If all of the resulting bits are '0,' the output of the OR gate is '0,' indicating that both hash addresses are identical. If at least one of the resulting bits from the exclusive-OR gates is '1,' then the output of the OR gate becomes '1,' signifying that the loaded hash address in the address register and the incoming hash address are different. Then the output ('1') of the OR gate triggers the K input of the JK flip-flop (The output of the JK flip-flop is initially cleared to be '1' by the controller.), so the output of the JK flip-flop becomes '0.' Therefore, the five structurally identical hash address comparators in the DBCP generate output signals at the same time.

The hash address comparator (HAC) has a second purpose. If the HAC is pushed into the stack, the hash address in the HAC is used to keep track of the next bucket address to be processed. Just before the HAC is pushed into the stack, the hash address register is cleared by the controller. The first hash address is, therefore, '0,' and the bucket zero is examined if it is empty. The inverted signal from the connected BAS tells whether or not both the source and target buckets are empty. If at least one of the buckets is empty, and if the controller allows it, the inverted signal ('1') increments the hash address register. This incrementing process is repeated until a pair of non-empty buckets is

found. Before the source and target tuples in those buckets are further processed, another pair of non-empty buckets is found and the bucket address is stored in the hash address register. This bucket address is stored in the stack for later use. As a result, when the HAC is stored in the stack, the associated hash address register is used to store the next non-empty bucket address.

The hardware for hash address comparison, required to detect whether all the join attributes in a file or list are identical, merits elucidation. The purpose of this hardware is to inform the controller whether or not the input file or list should be divided further. If so, the DBCP eventually sends the source, and target tuples having the same join attribute, to the host processor for concatenation. As shown in Figure 4-9, the five hash address comparators are stacked. Based on the value in the stack pointer register, the 5-to-1 multiplexer selects one from the five inputs. When the stack pointer designates the first, i.e. lowest, stack level, all the outputs from the HACs are ANDed, and the resulting output of the AND gate (A) is selected by the multiplexer. If the stack pointer specifies the second stack level, the first BAS is saved in the stack and is not written until the controller sends a memory write signal to the BAS. The output of the first HAC is, therefore, excluded from the inputs into the AND gate (B), and outputs of the second, third, fourth, and fifth HACs are ANDed.

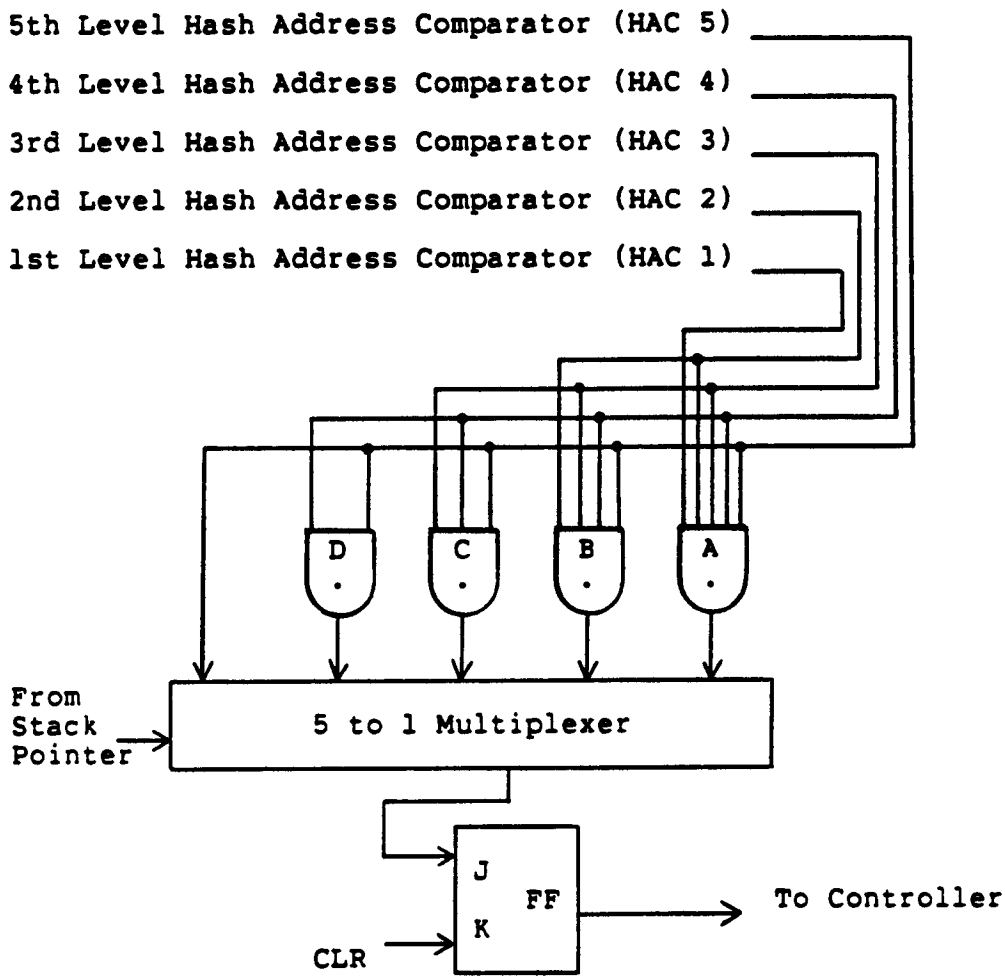


Figure 4-9. Condition Code for Checking if Only One Kind of Hash Address is Produced

Likewise, if the indicated stack level is the third level, the first and second BASs are saved in the stack and the multiplexer chooses the output of the AND gate (C), which receives the outputs from the third, fourth, and fifth HACs as inputs. If the indicated stack level is the fifth and

highest level, the four lower level BASs are saved and the multiplexer selects the output directly from the fifth level HAC.

The single bit output from the multiplexer triggers the attached JK flip-flop if, after a whole input file or list has been scanned, all the HACs, which are equal or higher than the current stack level, indicate that only one kind of hash address has been produced from each hash coder. The output value of the JK flip-flop is then sent to the controller. The controller, based on the value from the JK flip-flop, then decides either to continue a division process or to require a conquer process. In the conquer process, the controller sends the list of the tuples which are not filtered to the host processor for a merge.

Even though the output signal indicates that no further division process is necessary, there is approximately one chance in a trillion (1/256**5) that the signal will pass an unwanted key. The final screening with direct comparisons by the host processor will eliminate the spurious key, if it is present. Because this chance is extremely small, the host processor will not waste time dealing with unnecessary data.

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 group of hash address

comparators determines whether or not the scanned tuples have the same join attribute, and provides information to the controller concerning further division process or sending the desired tuples to the host processor.

The major operation in the filter unit is hashing for dividing and filtering tuples. A maximum five hash coders may participate in producing hash addresses in parallel. If this filter device (DBCP) is implemented as a software back-end, the hash addresses are calculated serially, one after another, using a slow software hash coder. As discussed, the mapping hash coder implemented in software produces a hash address about 32 times slower than the mapping hash coder implemented in hardware. Additionally, because the software back-end has to calculate five hash addresses in serial, it may be 160 (32*5) times slower than the hardware back-end in the computation of hash addresses.

Both the parallel architecture of the hardware back-end DBCP for the five hash coders and the parallel architecture of each hash coder can drastically reduce the execution time of the join. Since the software back-end cannot take advantage of the speed of parallel processing, it is recommended that the DBCP should be implemented as a hardware back-end.

# CHAPTER 5

# A NEW HASH-BASED JOIN ALGORITHM

In the previous chapter, the hashed address bit array store filtering is mentioned because the architecture of the DBCP is designed to support the filtering technique. The first section of this chapter contains additional discussion on the hashed bit array store filtering technique. 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; the results and statistics of this simulation are shown in the fourth section. Because other relational database operations, such as project, union, difference, and intersect, have hashing in their nature, the last section discusses how these operations also can utilize the hash coder in the DBCP effectively.

## 5.1 Limitation of Hashed Bit Array Store Technique

A problem called hashing collision, associated with the hashing technique, occurs when more than one key applied to the same hash function generates the same hash address. To minimize this problem more than one functionally different hash coder is used, as was explained in the previous chap-

ter. The major problem of the hashed address bit array store technique takes place when the source relation--smaller than the target relation--is very large. A one shot filtering scheme of the hashed address bit array store does not eliminate all of unwanted data; therefore, the host processor has to carry the burden. The following discussion shows how the problem of the hashed address bit array store filter technique arises in the DBCP architecture.

As shown in Figure 4-5, the DBCP has five different hash coders. Each key or join attribute value is transformed into five hash addresses by the five functionally different hash coders. Furthermore, the five hash addresses are used to mark single-bit wide source RAMs in the corresponding bit array stores (BASs). The chance that two different keys will have five pairs of identical hashed values is extremely small if five statistically independent mapping hash functions are used. However, the hashed bit array store technique on its own still allows for the existence of spurious keys. This problem can be explained using Figure 5-1, which shows that key K1 in a tuple of the source relation A 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 five hash functions to generate five

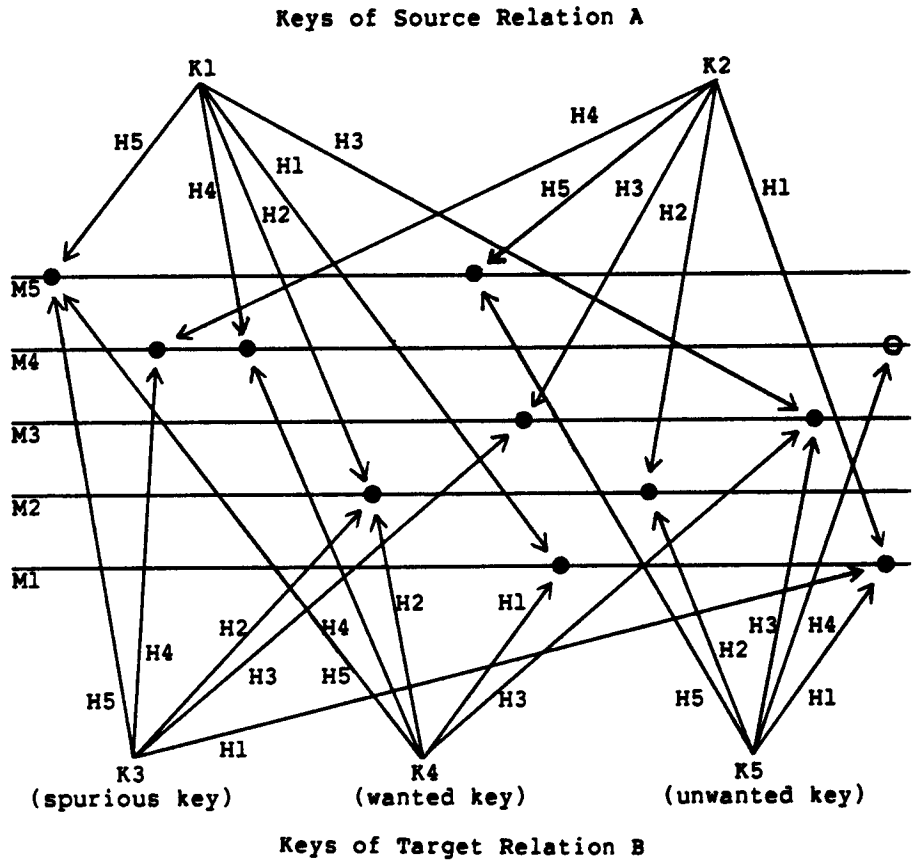


Figure 5-1. Key Mappings

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, K3 is regarded as a join attribute and matched with a join attribute of some

particular tuple in the source relation A. In reality, K3 is neither K1 nor K2, but is a spurious key. Key K4 is, on the contrary, a wanted key, and its associated tuple will be concatenated with the tuple of key K1 by the host processor after a final screening. Key K5 will be discarded as soon as the logical AND value of five bits from the source RAMs is detected to be zero due to M4(H4(K5)) = '0'.

There will be more spurious keys when many bits in M1, M2, M3, M4, and M5 source RAMs become '1' after a large size source relation has been scanned. Therefore, this problem is an inherent data-size dependent problem of the hashed address bit array store 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. As a result, the host processor has to spend time discarding spurious tuples by tedious key comparisons. The next section shows how HIMOD overcomes the spurious key problem using the stack oriented filter technique.

## 5.2 Stack Orient Filter Technique

As discussed in Chapter 4, the stack oriented filter technique (SOFT) is the main idea used in a new join algorithm. Internally, there is a stack containing five items, each of which is virtually a bit array store (BAS). At the

beginning, the lowest BAS is the item at the top of the stack. The stack pointer keeps track of the top item of the stack or the current BAS, since another BAS may be added onto the top of the stack and the current BAS may be deleted from the top of the stack.

Several primitive operations in the stack data structure, such as push, pop, and Bottom_Of_Stack, are provided for use in the new join algorithm. With five BASs, the upper limit in the stack is five. Therefore, it is not allowed to push another BAS (item) when five BASs are already stacked. The operation Bottom_Of_Stack indicates if the stack pointer points to the lowest BAS in the stack. In this situation, the pop operation cannot be applied to the stack because the stack in the SOFT keeps at least one BAS.

In the process of the join, there might be three different types of BAS (as shown in Figure 5-2): Current BAS, Saved BAS, and Available BAS for the hashed address bit array store filtering technique. The current BAS is the one that is pointed to by the stack pointer, and it is also involved in the hashed address bit array store filtering technique. The input tuples are stored in the addressed bucket by the hash coder, which is connected to the current BAS.

The source RAM in every BAS has 256 bits. And if the value of i-th bit of single-bit wide RAM in either current or saved BAS is '1,' the i-th bucket is not an empty bucket,

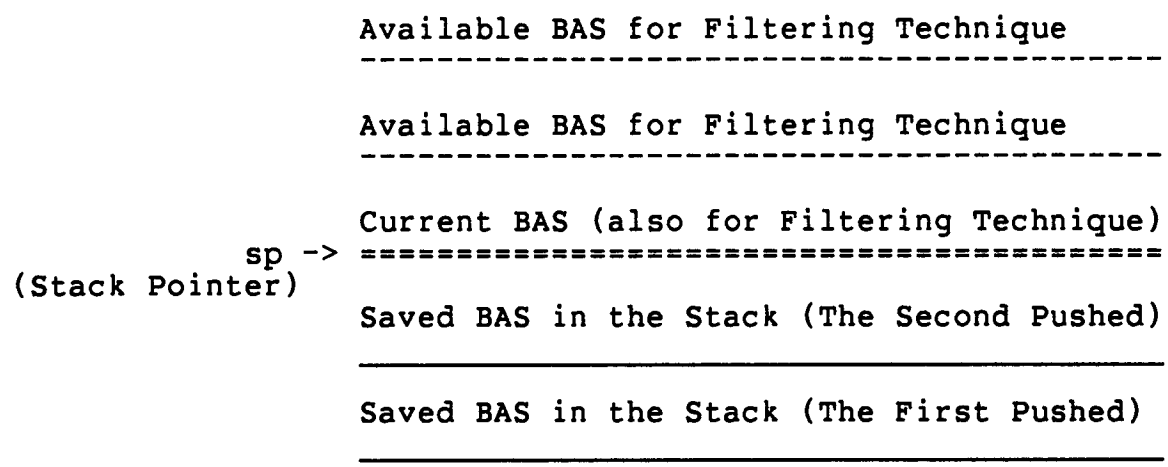


Figure 5-2. Stack Configuration after Two Items are Pushed

and its corresponding subset file or subset linked list contains collected tuples. On the other hand, if the value of the i-th bit is '0,' there has been no key in the target inputs addressed to this bucket by an affiliated hash coder; thus, no tuple is stored in this bucket. After the source and target relations are scanned, the lowest BAS has information for key distributions of these inputs. If both input relations are too large to fit into the main memory, they are divided into a maximum of 256 subset files for each relation by the associated hash coder of the lowest BAS, as is shown in step 1 in Figure 5-3.

If each BAS produces only one hash address in an entire scanning of keys, an output from the five hash address comparators sends a signal to the controller indicating that almost all the unnecessary tuples have been filtered, as was

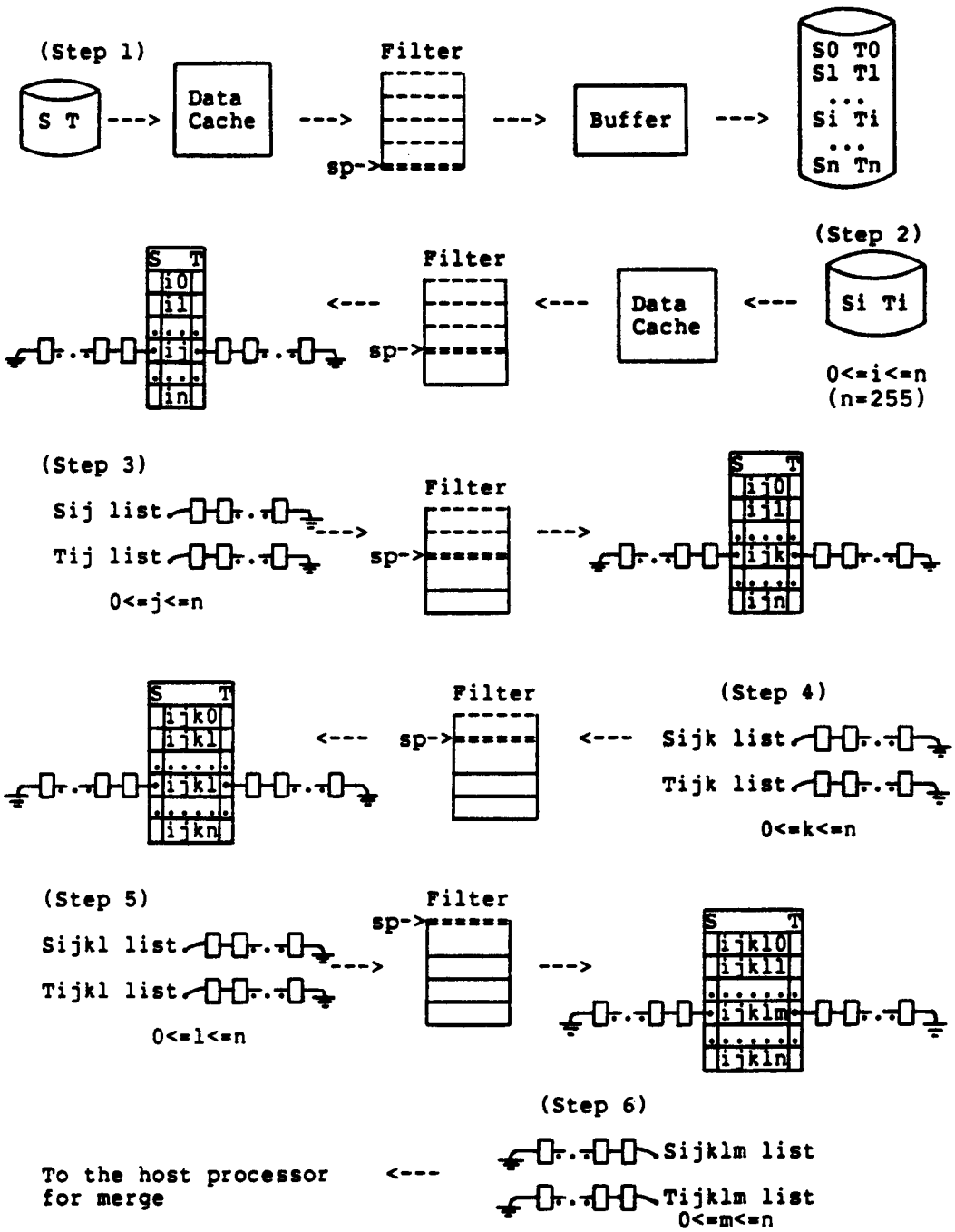


Figure 5-3. Join Process in the SOFT

discussed in Chapter 4. In this case, although extremely rare, all the tuples in the source and target subset files are sent to the host processor for merging. Otherwise, it is obvious that unnecessary or matchless tuples can be included in the i-th source and target subset files, since the SOFT filters those unnecessary tuples efficiently. It also is clear that the tuple(s) of the i-th source subset file might have matched tuple(s), if there are any, only in the i-th target subset file.

Next the i-th (0<=i<=n=255) nonempty source and target subset files will be divided again by the SOFT. However, the next subset files, after the i-th, should be saved along with the lowest BAS for subsequent processing. Since the current BAS is pushed onto the stack, 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 address bit array filter, are then stored in the hash-address buckets by the hash coder associated with the second BAS.

As shown in Step 2 of Figure 5-3, a hash table which has source(S) and target(T) pointers, to either a linked list or a null in each bucket, will be created. If each one of the four BASs used in the filtering process produces only one hash address in the entire scanning of keys, an output from the four hash address comparators sends a signal or

condition code to the controller. This condition code indicates that it is all right to send the tuples in a pair of source and target lists to the host processor for merging. The SOFT then returns to Step 1 in order to process the next subset file. It quits if no subset file remains.

If no signal comes from the four hash address comparators, the tuples in the ij-th (0<=j<=n=255) source and target linked lists (which has tuples) need to be divided again. The current BAS is pushed onto the stack; thus, 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 in the hash-addressed linked lists by the third hash coder, as was shown in Step 3 of the Figure 5-3.

Step 4 and Step 5 in Figure 5-3 can be explained similarly. In Step 6, no available BAS is left, and unnecessary data have been filtered; therefore, the source and target tuples are sent to the host processor without filtering.

As far as data structure is concerned, the linked list data structure is better than the array data structure for the buckets in steps 2 through 5 in Figure 5-3 because in these steps most of the buckets may be empty. Therefore, by using the linked lists for the buckets, memory space can be conserved.

The SOFT can be implemented in a recursive routine, so

that bit array stores are structured as elements of a stack. The nature of recursion in the SOFT simplifies the architectural structure of the DBCP in HIMOD.

## 5.3 The New Join Algorithm

The fundamental data structure used in the new join algorithm is a stack, as explained in the previous section, because the design of this join algorithm is based on the stack oriented filter technique (SOFT). Push and pop are the names of procedures operating in the stack: push inserts a bit array store (BAS) onto the top of the stack, and pop deletes another from the top. The stack pointer always points to the current BAS--the item at the top of the stack---by incrementing its value when pop is called. By referring to the value in the stack pointer, the function Bottom_Of_Stack can tell whether the stack pointer points to the first or lowest 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, No_More_Next_Bucket_Addr, and Save_Next_Bucket_Addr. The module Assign_Source_And_Target uses the header pointers of both source and target linked lists based on the saved next bucket address of the current BAS (or HAC) in order that the tuples in the linked lists are processed through the filter again. As explained in

Chapter 4, each BAS in the DBCP has a connected hash address register in the associated hash address comparator (HAC). Each next bucket address is saved in the hash address register and incremented to keep track of the next bucket address. Whenever the procedure Assign_Source_And_Target is called, another next bucket address, which has the value '1' for the resulting output of ANDed bits, from the source and target RAMs in the current BAS, is found by the procedure termed Save_Next_Bucket_Addr. This procedure also saves the next bucket address for the next process which is combined with the current BAS. Then the procedure push saves the contents of the current BAS and increments the stack pointer in order that the next upper BAS becomes the current BAS or top of the stack.

When pop is called, the stack pointer is decremented in order that the BAS directly under the current BAS becomes the current BAS. After the pop, the boolean function No_More_Next_Bucket_Addr should be called in order to see if there is any saved next bucket address in the hash address register in the current BAS--generally, and in the current HAC--specifically. If there are none, the current BAS is checked to see if it is the first or lowest BAS. If so, the join process is terminated by breaking the repeat loop.

The algorithm shown in Figure 5-4 is an explanatory version of the main module of the simulation program. After an appropriate initialization process (1) (including finish

:= true), the algorithm reads the two input files for source and target relation from the secondary storage (2). Then there is a large repeat loop (3) starting with clearing the current and upper available BAS(s) for use in the hashed address bit array store filtering technique (4). Next, the algorithm scans the tuples in the source linked list or (subset) file, by hashing their join attributes and setting up hash-addressed bits in the corresponding BAS (5). The source tuples are divided by the hash coder of the current BAS, and stored in the addressed bucket (linked list). Then the algorithm hashes the join attributes of the target tuples in the list; the algorithm then detects and eliminates unnecessary tuples by examining the ANDed result of all the BAS(s) involved in the hashed address bit array store filtering process (5). The target tuples, which are not filtered in this step (5), are saved in the buckets which are addressed by the hash coder associated with the current BAS.

There is an if-statement (6) testing if the produced hash addresses are all identical, by examining the condition code set by the hash address comparators as in HIMOD. If so, headers of source and target linked lists (or subset files) are sent for merge (8). The algorithm checks if there is a next bucket address (9). If so, it assigns headers of source and target linked lists in the next bucket address to

```
(0) begin
(1) Initialization;
(2) Form_Source_And_Target_Relations;
(3) repeat
(4)   Clear_Current_And_Upper_BASs;
(5)   Hash_Source_And_Target_Relations;
(6)   if Only_One_Hash_Address_Produced_In_Each_HAC then
(7)   begin
(8)     Merge_Tuples_And_Output;
(9)     if No_More_Next_Bucket_Addr then
(10)    begin
(11)      if Bottom_Of_Stack then
(12)        finish := true
(13)      else
(14)      begin
(15)        pop;
(16)        if No_More_Next_Bucket_Addr then
(17)        begin
(18)          if Bottom_Of_Stack then
(19)            finish := true
(20)          else
(21)          begin
(22)            pop;
(23)            if No_More_Next_Bucket_Addr then
(24)            begin
(25)              if Bottom_Of_Stack then
(26)                finish := true
(27)                else
(28)                begin
(29)                  pop;
(30)                    if No_More_Next_Bucket_Addr then
(31)                    begin
(32)                      if Bottom_Of_Stack then
(33)                        finish := true
(34)                      else
(35)                      begin
(36)                        pop;
(37)                        if No_More_Next_Bucket_Addr
(38)                        then begin
(39)                          if Bottom_Of_Stack then
(40)                            finish := true
(41)                            else
(42)                            begin
(43)                              Assign_Source_And_Target;
(44)                              Save_Next_Bucket_Addr;
(45)                              push;
(46)                            end;
```

Figure 5-4. The New Join Algorithm in Pascal

```
(47)                          end;
(48)                        end;
(49)                      end
(50)                      else
(51)                      begin
(52)                        Assign_Source_And_Target;
(53)                        Save_Next_Bucket_Addr;
(54)                        push;
(55)                      end;
(56)                    end;
(57)                  end
(58)                else
(59)                begin
(60)                  Assign_Source_And_Target;
(61)                  Save_Next_Bucket_Addr;
(62)                  push;
(63)                end;
(64)              end;
(65)            end
(66)          else
(67)          begin
(68)            Assign_Source_And_Target;
(69)            Save_Next_Bucket_Addr;
(70)            push;
(71)          end;
(72)        end;
(73)      end
(74)    else
(75)    begin
(76)      Assign_Source_And_Target;
(77)      Save_Next_Bucket_Addr;
(78)      push;
(79)    end;
(80)  end
(81)  else
(82)  begin
(83)    Assign_Source_And_Target;
(84)    Save_Next_Bucket_Addr;
(85)    push;
(86)  end
(87) until finish;
(88) end.
```

Figure 5-4. Continued

two temporary pointer variables for the filtering process (76). Then the next bucket address--in the next address reg-

ister of the current BAS in HIMOD--is incremented until another bucket address which is '1' in its content in the current BAS is found (77). If no such next bucket address is found, it assigns a null value to the next bucket address. After that, it pushes the current BAS onto the stack, and goes back to the beginning step of the repeat loop, starting with clearing the current and available BASs (4), and then hashes the source and target tuples in the assigned lists in the line 76 (5).

In line 9, if there is no next bucket address, then the algorithm checks to determine if the current BAS is the first (lowest) BAS (11). If so, it assigns a true value to the repeat condition variable finish (12), and the join process is terminated. If a BAS(s) is saved, then the algorithm pops the BAS (15) and checks if there is a saved next bucket address which is not a null value (16). If there is a next bucket address, then the algorithm performs the same sequence as explained for lines 76, 77, and 78 (68, 69, and 70) using the next bucket address as the bucket address of the current BAS. If there is no next bucket address, the algorithm either pops the stack (22) or assigns a true value to the variable finish (19) if the current BAS is the lowest item of the stack (18).

There are several nested if-then-else statements that include the same code patterns. Accordingly, this algorithm also can be implemented using a recursive routine. Since the

stack can have only five items, maximum, this non-recursive algorithm is useful and can be implemented easily in the controller (ROM) of the DBCP in HIMOD.

## 5.4 Simulation Results:

### A Comparison with the Conventional Join

The simulation has been performed on an IBM 4381 mainframe computer. The listing of the simulation program in Pascal is outlined in the Appendix. The set resulting from the combination of the 1,024 generally chosen names-data set and the 1,024 randomly chosen names-data set are 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 contains 2,048 name tuples, which is read into the system in order to create both the source relation and target relation. As each tuple is scanned, the initial letter of the last name is compared to the discriminator character variable. For example, if the discriminator is set to be "K," then the name tuples whose last name initials are from "A" to "K," are inserted into the source relation, while all others will be inserted into the target relation, e.g., "L" to "Z." For each name, the last name is used as a join conditional attribute, and the hash address is calculated using only the last name. The whole name is used to produce a hash address in the hash algorithm experiments. After creating

the source and target relations, the equi-join operation is performed on those input relations in order to produce the resulting relation for the join. The source, target, and correct resulting relations are printed as proof that the algorithm logically works.

As shown in Table 5-1, the number of tuples brought into the processor is selected as the major measurement of overall performance, although there are other factors to be considered, such as the number of disk accesses and I/O time. Since more data movements might create frequent disk accesses, which will in turn slow the join operation, the fewer number of tuples brought into the CPU, the shorter the response time will take for the join.

| | Number of tuples in relations | | | No. of tuples brought into the Processor | |
|---|---|---|---|---|---|
| Discri-minator | Source | Target | Resulting | New Join Using the SOFT | Nested-Loop Join |
| A | 155 | 1,893 | 355 | 2,426 | 293,415 |
| E | 646 | 1,402 | 919 | 3,795 | 905,692 |
| G | 799 | 1,249 | 902 | 4,095 | 997,951 |
| K | 1,196 | 852 | 846 | 4,419 | 1,018,992 |
| V | 1,961 | 87 | 205 | 2,902 | 170,607 |
| | | | Sums: | 17,637 | 3,386,657 |

Table 5-1. Number of Tuples Brought into the Processor

In this particular case, on average, the new hash-based join using the SOFT takes about two hundred times (3386657/17637) less data movements than the conventional nested-loop join method. The main reason for this contrast in performance is that the SOFT eliminates all unnecessary data--about 99.9999999999%--in the filtering process while dividing the source and target relations into groups of tuples so that source tuples in a group will be matched only with those target tuples in a corresponding group. Removing unnecessary data, while hashing and dividing, helps reduce data movements drastically. The conventional nested-loop and sort-merge, on the contrary, carry around every tuple--even if most of them are not wanted--all the way to the last moment before they discover it is an unnecessary tuple through direct comparison of join attributes.

The time complexity of the new join algorithm is O(S+T+R) where the number of tuples in the source relation relation is S, the number of tuples in the target relation is T, and the number of tuples in the resulting relation is R. This time complexity is actually the same with those of other hash-based join methods, such as simple, GRACE, and Hybrid. However, the new hash-based join method will outperform the aforementioned hash-based join methods because none of them includes the concept of filtering in their algorithms. Their common method--finding a matched source tuple for an incoming target tuple--entails comparing the join

attribute of the target tuple with every source join attribute in the hash-addressed bucket, one by one. This repeated direct comparison slows the system; however, in the new join method, the direct comparison of the join attributes is only allowed after all of unwanted data are cast out.

## 5.5 Other Relational Operations Which Utilize a Hash Coder

### 5.5.1 Project (Eliminating Duplicates)

After the project relational database operation selects only those attributes, specified in the attribute list, from each tuple in the input relation, there may be duplicates in the resulting relation. For example, in Figure 5-5, there is 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 | n | m |
| a | d | i | h |
| j | e | c | i |

PROJECT R (B,D)

| B | D |
|---|---|
| e | k |
| d | h |
| c | m |
| d | h |
| e | i |

--->

| B | D |
|---|---|
| e | k |
| d | h |
| c | m |
| e | i |

Figure 5-5. Example of Project Operation

In the intermittent resulting relation, the second and the fourth tuples have the same values (d and h) for the

attributes B and D, and therefore, duplicate tuples, like the fourth tuple, have to be eliminated.

DBMSs generally sort the tuples in the intermittent resulting relation in alphanumerical order. They search and eliminate unnecessary tuples. To find duplicate tuples in the sorted relation, two pointers are usually employed 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 tuple pointed by the second pointer. During the comparison process, both of the pointers are incremented, whenever necessary, until they reach the last tuple in the sorted relation.

In HIMOD, hashing is used instead of sorting for the project operation by taking advantage of the fast hash coders in the DBCP. The partial or partial combined value of attributes in the attribute list--or a folded tuple is used as a key if a tuple is too long--is hashed by the five statistically independent hash functions. The HIMOD uses the five produced hash addresses to examine the hash-addressed bits in the corresponding BASs. If at least one bit in any BAS is not set, then no identical tuple has passed through the filter. Thus, the input tuple is stored in a bucket based on the hash address produced by the first hash coder. In this case, it is not necessary to compare the tuple with previously stored tuples in the hash-addressed bucket. Next 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 possible to have an identical tuple that has already passed through the filter. Therefore, the whole input tuple must be compared to the tuples in the hash-addressed bucket indicated by the first hash coder. If there is a match, the tuple is instantly eliminated. Then the 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 derived from using the DBCP is that it determines the necessity of tuple comparisons from the beginning by examining the bits in the five BASs. As a result, time for comparisons may be, to some extent, saved.

When the intermittent resulting relation does not fit in the real memory, the relation should be divided into subset files based on the hash address produced by the first hash coder. Depending on how the main memory is used (e.g., as a hash table, output buffers, and both output buffers and a hash table), the method for project operation might be different.

Once the intermittent resulting relation is divided into subset files, tuples in a subset file are 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 discarded. After one subset file is finished, another subset file is processed until no subset files remain.

### 5.5.2 Union

The hash coder in the DBCP can be utilized efficiently in performing the UNION operation (U). As shown in Figure 5-6, the source relation (R1) is first read from the secondary storage. If the source relation R1 does not fit in main memory, the tuples--a folded tuple or some portion of a tuple is used for a key if a tuple is too long--in R1 should be divided into subset files (S0, S1, ... , Sn-1) by the first hash coder. The target relation R2 is also divided into subset files (T0, T1, ... , Tn-1) by the same hash coder. After the division process, the tuples in the first source subset file (S0) are hashed by the five hash coders in the DBCP. The hash-addressed bits in the corresponding BASs are set, and the tuples are stored in the hash-addressed bucket in main memory. Then the tuples in the first target subset file (T0) are hashed by the same five hash coders in the DBCP, and the five bits in the five BASs are examined to detect the probability that the same tuple already has passed through. If all five bits are set, it is probable that an identical tuple is in the S0. Thus the hash-addressed bucket is searched for a match by the first

hash coder. In this process, the whole tuple is compared as a key. If there is a match, the target tuple, e.g., 5 a and 1 b in Figure 5-6, is eliminated immediately. Otherwise, the target tuple should be included in the resulting relation.

| Relation R1 | | Relation R2 | | R1 U R2 | |
|---|---|---|---|---|---|
| 5 | a | 5 | a | 5 | a |
| 3 | a | 10 | b | 3 | a |
| 9 | a | 15 | c | 9 | a |
| 1 | b | 2 | d | 1 | b |
| 2 | b | 6 | a | 2 | b |
| 4 | b | 1 | b | 4 | b |
| | | | | 10 | b |
| | | | | 15 | c |
| | | | | 2 | d |
| | | | | 6 | a |

Figure 5-6. Example of Union Operation

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 (S1) and target (T1) relations. The same process is repeated for the second subset files, and all subsequent files until all files have been processed. If there is sufficient main memory space, it is not necessary to divide the relations; in this case, the same process for each subset file is applied to the whole input file that resides in main memory.

### 5.5.3 Difference

The executional process of difference relational database operations (-) approximates that of the union operation discussed in the previous section. As shown in Figure 5-7, the source relation R1 is first read from the secondary storage. If the source relation does not fit in main memory, it should be divided into subset files (S0, S1, ... , Sn-1) by the first hash coder. By the same hash coder, the target relation R2 is also divided into subset files (T0, T1, ... , Tn-1). After the division process, the tuples in the first source subset file (S0) are hashed by the five hash coders in the DBCP. Some portion of a tuple or a folded tuple is used for a key if a tuple is too long. The hash-addressed bits in the corresponding BASs are set, and the tuples are stored in the hash-addressed bucket in main memory. The tuples in the first target subset file (T0) are hashed by the same five hash coders in the DBCP; the five bits in the five BASs are then examined to test the possibility that the same tuple has already passed through. If all five bits are set, an identical tuple in the S0 is probable. In such an instance, the hash-addressed bucket by the first hash coder is probed for a match. In this process, the whole tuple is compared as a key. If a duplicate is found, e.g., 5 a and 1 b tuples in Figure 5-7, both source and target tuples are eliminated from the addressed-bucket. Otherwise, the target

tuple is stored in the hash-addressed bucket. Eventually, the tuples remaining in the hash table make up a resulting relation.

| Relation R1 | | Relation R2 | | R1 - R2 | |
|---|---|---|---|---|---|
| 5 | a | 5 | a | 3 | a |
| 3 | a | 10 | b | 9 | a |
| 9 | a | 15 | c | 2 | b |
| 1 | b | 2 | d | 4 | b |
| 2 | b | 6 | a | | |
| 4 | b | 1 | b | | |

Figure 5-7. Example of Difference Operation

### 5.5.4 Intersect

The executional process of the intersect operation (&) is similar to the join operation. A portion of an input tuple or folded form of a tuple is considered to be a key if the tuple is too long. If the source relation does not fit into the main memory, it must be divided into subset files by using buffers based on the hash addresses resulting from the first hash coder. Every tuple in the source subset file (S0) is hashed and the hash-address bits in the five BASs are marked. Then the tuples in a target subset file (T0) are hashed, which causes an examination of the corresponding bits in the BASs. If at least one bit is not set, the target tuple has no potential to be included in the resulting relation; thus, it is eliminated immediately. Otherwise, the

hash-addressed bucket is probed for a match. If there is a match, the tuple is included in the resulting relation, e.g., 5 a and 1 b tuples in Figure 5-8.

| Relation R1 | | Relation R2 | | R1 & R2 | |
|---|---|---|---|---|---|
| 5 | a | 5 | a | 5 | a |
| 3 | a | 10 | b | 1 | b |
| 9 | a | 15 | c | | |
| 1 | b | 2 | d | | |
| 2 | b | 6 | a | | |
| 4 | b | 1 | b | | |

Figure 5-8. Example of Intersect Operation

After hashing tuples in the first pair of subset files (S0 and T0), the five BASs are cleared and all the source tuples in the hash table are eliminated. The same procedure is repeated for the second pair of subset files (S1 and T1) and for all subsequent pairs. The matched tuples are accumulated in the resulting relation during those processes.

# CHAPTER 6

## SUMMARY AND CONCLUSIONS

The previous sections addressed the necessity of database computer development since conventional computers are not inherently optimized for nonnumeric processing. Due to the advantages of the relational data model, the relational data model has been chosen for most of the database machines. The major problem with relational database machines, however, develops from the frequently used and time-consuming join operation. Thus, it is apparent that accelerating the join operation will improve the performance of relational database systems.

As described in Chapter 2, 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 in many database computers during the early stages of database machine development. After Babb's hashed bit array stores <BABB1> was introduced, researchers began recognizing the importance of the filtering technique <DEWI3, QADA2, SHAP1, VALD2>. Combining hashing and filtering techniques would appear to be an ideal approach. Although Goodman <GOOD1> and several other researchers have taken advantage of Babb's hashed bit array

stores filtering technique, the ideal filtering process has not yet been fully explored.

This dissertation outlines an ideal approach for filtering, and discusses the highly modular relational database computer, HIMOD, equipped with a single chip back-end processor for the join operation. The parallel multiprocessing was not chosen for this study due to its complex synchronization problems and lack of cost effectiveness. However, in future research it would not be excluded from the study. In the course of this dissertation research, a single join back-end processor with specialized hardware which maximizes the filtering effect during the hashing process has been developed.

The join database back-end processor (DBCP) is a stack oriented filter device. Five statistically independent hash coders within the DBCP have an associated bit array store (BAS) and an associated hash address comparator (HAC). The stack pointer always points to the current BAS. The BASs that reside below the current BAS are saved in the stack for later use as marks for empty and non-empty buckets. The current BAS, and the BASs above the current one, participate in the hashed address bit array store filtering process. The hash coder, 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), dis-

cussed in Chapter 5, is applied using the five BASs in the stack in three different ways (e.g., current BAS, saved BAS, and available BAS) for filtering technique. My development of new hash-based join algorithm is based on this technique.

The distinguishable difference between the new join algorithm and other hash-based join algorithms (such as straightforward, simple, GRACE, and Hybrid) is that, in the new hash-based join algorithm, the filtering process is combined with the hashing process. Accordingly, unnecessary data are detected and filtered by the hashed address bit array store filtering technique while other join algorithms carry unwanted tuples up to the last moment of join attribute comparisons.

The new hash-based join algorithm repeats this division and filtering process many times in a recursive way; therefore, nearly 100 percent of unnecessary tuples are filtered. The tuples that are left after the filtering process are sent to the host processor for final screening and merging. The transmitted source and target tuples are then merged in order to produce an output for resulting relation. After repetitive division and filtering processes, the remaining tuples in the source and corresponding target list have an extremely high probability of having identical join attributes. All other tuples are eliminated before unnecessary comparison of their join attributes begins. This elimination of unnecessary tuples substantially reduces the number

of join attribute comparisons. As a result, total data movements in performing a join are radically diminished.

Although the CAFS's hashed address bit array store filtering technique is adapted by the HIMOD, use of the CAFS filter device, which resides between the secondary storage and the main memory, would still improve the overall system performance. If CAFS dramatically improve all join algorithms <DEWI3, QADA2, SHAP1, VALD2, SCHN1>, then adding the CAFS I/O filter to the HIMOD database computer would result in an even faster relational join. Thus, this addition of the CAFS I/O filter is strongly recommended.

The output of the join simulation program shows that the new hash-based join algorithm is logically correct. Furthermore, the number of tuples passed through the processor in the simulation manifests how effectively the new join method has cut down data movements as compared to conventional methods.

The new join method can be divided into two processes: the filtering process and the merging process, with final screening. In the HIMOD database computer, the filtering process is performed by the database coprocessor (DBCP), and the merging process is executed by the host processor whenever it receives source and target lists of tuples from the DBCP.

The HIMOD uses a Motorola 68030 microprocessor (MC68030) as the host processor, and the DBCP communicates

with the host processor through a protocol defined as the M68000 coprocessor interface <MOTO1>. The DBCP may be designed as either a software back-end or a hardware back-end. For a software back-end, the MC68030 can be used and the filtering process of the new join algorithm is thus implemented in software. In order to save the cost of developing a new coprocessor, one must sacrifice the speed of the join because the software back-end takes about 160 times longer in the filtering process than the newly designed hardware back-end.

The architecture of the hardware back-end has been illustrated in Chapter 4. The illustration outlines the design of the stack oriented filter device which efficiently filters unwanted data. The major operation of the DBCP in filtering data is hashing: five hash coders in parallel produce five hash addresses. Each hash function should be statistically independent; therefore, it is important that the five hash addresses from a key are not related to each other.

In Buchholz <BUCH1> and Lum's <LUM1> review of hash functions, they recommended the division method as the best. Knuth later concluded that none of the hash methods has proved to be superior to simple division and multiplication methods <KNUT1>. This statement is generally accepted as true in circumstances where distribution performance is essential and hash address calculation time is not.

Because huge amounts of data must pass through the hash coder in the DBCP, the hash address calculation in the DBCP should be very fast. In order to speed up the hash address computation, efforts should be concentrated on designing a new hash function that will avoid time-consuming serial and/or iterative computations while taking advantage of parallel processing, by means of hardware, for converting a key into a hash address. Moreover, the new hash algorithm should distribute random keys into buckets as uniformly as possible. The ideal hash function design for this database application is data-independent, and calculates a hash address within a few machine cycles with relatively good distribution. The new mapping hash method which has been outlined in this dissertation, is one that satisfies these requirements if it is implemented in hardware. The mapping hash method involves the combination of the mapping or converting of each character in a key to a corresponding prime-number or random-number technique and the folding technique.

Most of the well known hash functions, and several new ones, including mapping, shift-fold-encoding, Hu-Tucker code, and new versions of fold-shifting, are surveyed in this dissertation. Any necessary hardware aids are supplied during the implementation of the hashing process in order to increase the speed of the address calculation. Each hash function has been simulated and applied to two different name data sets (RCN and GCN) and one numeric string data set

(RNS) to produce distribution performances measured in terms of mean square deviations. The speed of calculating a hash address is measured in terms of clock cycles for each hash function in both the hardware and software implementation cases. The cost of the hardware implemented hash coder may be calculated and stated in terms of the number of gates used.

As the results illustrated in Table 3-1 indicate, the mapping hash method satisfies all three requirements at the highest rank. Moreover, this strategy also has advantages in generating statistically independent hash addresses during the same period of address calculation time by designating different sets of prime or random numbers for each hash coder. The Fold-shifting hash methods such as FS(0,10,20,30) and FS(0,11,22,25) have problems in finding three more similar types of statistically independent hash functions with relatively good distribution; nonetheless, they are fast and inexpensive. Both Maurer and Berkovich present new hash methods that have proved to be proficient in distribution performance. Their methods, however, have not been chosen for this specific application due to their relatively slow hash address calculation speeds.

This dissertation research has produced the development of a new join algorithm and new hash functions. This new join algorithm will shorten the time needed for a join, since it goes through frequent filtering processes to dis-

card unnecessary data efficiently. In turn, the more main memory space that is available, the more powerful this join method will be. The new mapping hash function will help speed the hash-based join algorithm and other hashing applications because parallel processing is used in the hardware hash coder in order to calculate a hash address in three clock cycles. Furthermore, the mapping hash method distributes keys effectively, compared to other well-known methods. The mapping hash method is also sensitive to every character in a key producing a hash address; that is, it does not have a data dependency problem in its distribution of similar keys. The author's mapping hash method is thus recommended for a hash coder in the join database coprocessor and in similar applications.

For future research, a database computer with multiple back-ends using the SOFT would be a fruitful research topic. The SOFT has an inherent characteristic of parallel processing. As shown in step 1 and step 2 of Figure 5-3, a single back-end processor can detect and eliminate unnecessary tuples in only one pair of subset files at a time. If two or more identical back-ends with local memory are provided, those subset files are processed in parallel. Thus, if the parallel processing is developed, then the speed of the join may be increased in proportion to the number of the back-ends used. This multiple back-ends database computer would outperform the multiprocessor database machines by using

well known join methods, such as parallel nested-loops join, parallel sort-merge join, and parallel hash-based join methods, since none of these methods exploits the filtering concept in their parallel join algorithms. A comparative study of these parallel join methods, including the HIMOD hash join, based on the measured response time, would provide a good solution for increasing the speed of the join.

# Appendix

## The New Join Algorithm

## Simulation Program Listing

```
{*********************************************************
 * This is the simulation program for the new hash-based
 * join algorithm. The main part of this program explains
 * the algorithm of the new join method.
 *********************************************************}
program Sim;

 const
     MAX_NUM_KEYS = 2048; {Total number of input keys}
     NUM_IDENT_CHAR = 16; {Number of characters in a key}
     MAX_BUCKET_ADDR_BITS = 8; {No. of bits in a hash addr}
     MAX_BOOL_DIGIT = 13; {No. bits for a prime number}
     NUM_ASCII_CHAR = 70; {No. of identifiable ASCII chars}

     NUM_FIRST_EXOR = 8;  {No. of gates in EX-OR module}
     NUM_SECOND_EXOR = 4;
     NUM_THIRD_EXOR = 2;

     BUCKET_SIZE = 255; {No. of buckets in the hash table}

     OUTPUT_FLAG = true; {Debug flags}
     SIM_DEBUG  = false;
     S_DEBUGED  = false;

     MAX_STACK_SIZE = 5; {No. of items (BASs) in the stack}
     NUL = 999;
     EMPTY = -1;
     DISCRIMINATOR = 'B'; {Discriminator variable}


 type
     {Type for key array}
     KEY_ARRAY_TYPE = array (.1..NUM_IDENT_CHAR.) of char;

     {Binary number type}
     PRIME_BOOL_TYPE = array (.1..MAX_BOOL_DIGIT.) of
                                                  boolean;
     {record for alphanumeric and other punctuation}
     CHAR_RECORD_TYPE = record
                          ch : char;
                          ASCII_num : integer;
                          prime_num : integer;
```

```
                    bool_prime : PRIME_BOOL_TYPE;
                end;

{ASCII lookup table}
ASCII_TABLE = array (.1..NUM_IDENT_CHAR,
                     1..NUM_ASCII_CHAR.) of
                      CHAR_RECORD_TYPE;

{Bit informations for each ASCII character in a key}
BOOL_PRIME_KEY_TYPE = array (.1..NUM_IDENT_CHAR.) of
                      record
                           bool_prime :
                           PRIME_BOOL_TYPE;
                      end;
{Stack type to keep keys in bits form}
STACK_PRIME_BOOL_TYPE = array (.1..MAX_STACK_SIZE.) of
                        record
                             Bool_Key_Arr :
                             BOOL_PRIME_KEY_TYPE;
                        end;

{Types for EX-OR module}
FIRST_EXOR_ARR_TYPE = array(.1..NUM_FIRST_EXOR.) of
                                              boolean;
SECOND_EXOR_ARR_TYPE = array(.1..NUM_SECOND_EXOR.) of
                                              boolean;
THIRD_EXOR_ARR_TYPE = array(.1..NUM_THIRD_EXOR.) of
                                              boolean;
HASHED_KEY_REG_TYPE = array(.1..MAX_BUCKET_ADDR_BITS.)
                                            of boolean;
{Type to store a hash address}
HASH_ADDR_TYPE = array (.1..MAX_BOOL_DIGIT.) of
                                              boolean;
{Stack element type for hash address}
CODER_TYPE = record
                Hashed_Addr : HASH_ADDR_TYPE;
             end;

{Stack type to keep hash addresses}
HASH_CODER_STACK = array (.1..MAX_STACK_SIZE.) of
                                            CODER_TYPE;
{Record type for the name data}
LINK = @KEY_RECORD;
KEY_RECORD = record
                Key_Arr : KEY_ARRAY_TYPE;
                First_Name_Arr : KEY_ARRAY_TYPE;
                next : LINK;
             end;

{Pointer array type for bucket pointers}
BUCKET_POINTER_ARRAY = array (.0..BUCKET_SIZE.) of
```

```
                                                    LINK;
    {Type for Bit Array Store (BAS)}
    BIT_ARR_TYPE = array (.0..BUCKET_SIZE.) of boolean;

    {Data Structure for each BAS--BAS, next address reg,
     and pointers for source and target buckets}
    BIT_ARR_RECORD = record
                       Bit_Arr : BIT_ARR_TYPE;
                       Next_Bit : integer;
                       Source_Bucket,
                       Target_Bucket :
                                   BUCKET_POINTER_ARRAY;
                     end;

    {Stack type for the BASs}
    BIT_ARR_TABLE = array (.1..MAX_STACK_SIZE.) of
                                     BIT_ARR_RECORD;

    {Next bit address}
    ADDR_TYPE_ARRAY = array (.1..MAX_STACK_SIZE.) of
                                                integer;
    {Record type for ASCII information}
    ASCII_RECORD = record
                     ASCII_Arr : ASCII_TABLE;
                   end;

    {ASCII stack type}
    TYPE_ASCII_STACK = array (.1..MAX_STACK_SIZE.) of
                                           ASCII_RECORD;
    {Types for statistics}
    REC_ELIMINATE = record
                       Total_Source, Total_Target,
                       D_Source, D_Target : integer;
                    end;

    ELIMINATE_TYPE = array (.1..MAX_STACK_SIZE.) of
                                      REC_ELIMINATE;

var

    ASCII_Arr : ASCII_TABLE; {Table for ASCII characters}

    Key_Char : char; {Variable for a character in a key}

    Bool_Stack : STACK_PRIME_BOOL_TYPE;

    EXOR1_Arr : FIRST_EXOR_ARR_TYPE;
    EXOR2_Arr : SECOND_EXOR_ARR_TYPE;
    EXOR3_Arr : THIRD_EXOR_ARR_TYPE;

    Code_Stack : HASH_CODER_STACK; {Stack for hash coders}
```

```
    Stk_Pt : integer;  {Stack pointer variable}

    {Temporary pointers for target and source linked lists}
    Hd_Source_Pt, Hd_Target_Pt : LINK;

    stack : BIT_ARR_TABLE; {Stack for bucket pointers (BAS)
                            and next bit address}
    finish : boolean;
    Addr_Num : integer;
    Hash_Value: integer;

    {Variables to count number of tuples in relations}
    Source_Count, Target_Count, Result_Count, Hash_Count :
                                                integer;
    ASCII_Stack : TYPE_ASCII_STACK;

    E_Count : ELIMINATE_TYPE;

    ASCII : text; {Identifiable ASCII characters and
                   Internal representation for chars}
    ROM_Nums : text; {Prime numbers file for ROMs}
    Names : text; {2048 names data set}
    out : text;   {output file}


{*********************************************************
 * This procedure Int_To_Bool_Convert converts integer
 * input number to a binary number.
 *********************************************************}

procedure Int_To_Bool_Convert (number:integer; var Bool_Arr:
                                              PRIME_BOOL_TYPE);

 var  i : integer;

 begin

    for i := 1 to MAX_BOOL_DIGIT do
         Bool_Arr(.i.) := false;

    i := 1;
    while (number >= 2) and (i <= MAX_BOOL_DIGIT) do
      begin

        if (number mod 2) =1 then
             Bool_Arr(.i.) := true
        else
             Bool_Arr(.i.) := false;

        number := number div 2;
```

```
          i := i + 1;
        end;

      if (number = 1) and (i <= MAX_BOOL_DIGIT) then
          Bool_Arr(.i.) := true;

end;


{************************************************************
 * Procedure Initialization initialize mainly by reading
 * inputs and storing them on the declared data structures
 ************************************************************}

procedure Initialization;

const
    char_divisor = 20;
    number_divisor = 10;

var  i, j, k, n : integer;
     number : integer;
     character : char;

begin

    {Reset input files.}
    reset (ASCII);
    reset (ROM_Nums);
    reset (Names);
    rewrite (out);

    for i := 1 to MAX_STACK_SIZE do
      begin
         with E_Count(.i.) do
           begin
              D_Source := 0;
              D_Target := 0;
              Total_Source := 0;
              Total_Target := 0;
           end;
      end;

{Assign null values for the ASCII table}

 for n := 1 to MAX_STACK_SIZE do
   begin

    with ASCII_Stack(.n.) do
      begin
```

```
        for i := 1 to NUM_IDENT_CHAR do
          begin

             for j := 1 to NUM_ASCII_CHAR do
               begin

                  with ASCII_Arr(.i,j.) do
                    begin
                       ch := '?';
                       prime_num := NUL;
                       ASCII_num := NUL;

                       for k := 1 to MAX_BOOL_DIGIT do
                            bool_prime(.k.) := false;
                    end;
               end;

          end;

   end;
end;

   {Then read inputs and store them in the ASCII table}

   for j := 1 to NUM_ASCII_CHAR do
     begin
        read(ASCII, character);

        for i := 1 to NUM_IDENT_CHAR do
          begin
             for k := 1 to MAX_STACK_SIZE do
               begin
                  with ASCII_Stack(.k.) do
                  ASCII_Arr(.i,j.).ch := character;
               end;
          end;

        if j mod char_divisor = 0 then
             readln(ASCII);
   end;

   readln(ASCII);

   {If the input number is too large for the table item,
    subtract 64 from the number before storing in table}

   for j :=1 to NUM_ASCII_CHAR do
     begin
        read(ASCII, number);
        if number > 64 then number := number - 64;
```

```
          for i := 1 to NUM_IDENT_CHAR do
            begin
               for k := 1 to MAX_STACK_SIZE do
                 begin
                    with ASCII_Stack(.k.) do
                    ASCII_Arr(.i,j.).ASCII_num := number;
                 end;
            end;

          if j mod number_divisor = 0 then
               readln(ASCII);

       end;

{Read prime numbers and store them into ROM in binary form}
 for i := 1 to NUM_IDENT_CHAR do
   begin

    for j := 1 to NUM_ASCII_CHAR do
      begin

         with ASCII_Stack(.1.).ASCII_Arr(.i, j.) do
           begin

              read(ROM_Nums, prime_num);

              if j mod number_divisor = 0 then
                   readln(ROM_Nums);

              Int_To_Bool_Convert(prime_num,
                                  bool_prime);

           end;  {with}
      end;  {for}

    readln(ROM_Nums);
    end;    {for}

    for n := 2 to MAX_STACK_SIZE do
      begin
         with ASCII_STACK(.n.) do
           begin
              for j := 1 to NUM_ASCII_CHAR do
                begin
                   with ASCII_Arr(.1,j.) do
                     begin
                        read(ROM_Nums, prime_num);
                        if j mod number_divisor = 0 then
                             readln(ROM_Nums);
                        Int_To_Bool_Convert(prime_num,
```

```
                                                    bool_prime);
                        end;
                    end;

                readln(ROM_Nums);

                for i := 1 to NUM_IDENT_CHAR do
                  begin
                     for j := 1 to NUM_ASCII_CHAR do
                       begin
                         ASCII_Arr(.i,j.).prime_num :=
                                 ASCII_Arr(.1,j.).prime_num;

                         for k := 1 to MAX_BOOL_DIGIT do
                            ASCII_Arr(.i,j.).bool_prime(.k.)
                         := ASCII_Arr(.1,j.).bool_prime(.k.);
                       end;
                  end;
              end;  {with}
        end;     {for}


    Stk_Pt := 1;
    finish := false;
    Addr_Num := NUL;

{Assign null to all the bucket pointers in the stack}

    for i := 1 to MAX_STACK_SIZE do
      begin
         with stack(.i.) do
           begin
              for j := 0 to BUCKET_SIZE do
                begin
                   Bit_Arr(.j.) := false;
                   Source_Bucket(.j.) := nil;
                   Target_Bucket(.j.) := nil;
                end;
             Next_Bit := NUL;
           end;  {with}
      end;   {for}

    Hd_Source_Pt := nil;
    Hd_Target_Pt := nil;

    Hash_Count := 0;
    Source_Count := 0;
    Target_Count := 0;
    Result_Count := 0;

end;
```

```
{*********************************************************
 * Procedure Form_Source_And_Target_Relations includes
 * several subroutines and reads input names to create
 * source and target relations comparing initial of each
 * last name with a discriminator variable.
 *********************************************************}

procedure Form_Source_And_Target_Relations;

var
    Key_Pt : LINK;
    Char_No_First, Char_No : integer;
    Source_Target_Flag, Target_Flag, Last_Name_Flag :
                                          boolean;
    Key_No : integer;

    {For Debugging Purpose}
    k1, k2 : integer;
    pt1, pt2 : LINK;



{*********************************************************
 * Procedure Init_While_Do_Loop does initialization
 * steps for next while do loop.
 *********************************************************}

procedure Init_While_Do_Loop;

var  i: integer;

begin

    Char_No := 1;
    new(Key_Pt);

    for i := 1 to NUM_IDENT_CHAR do
      begin
         Key_Pt@.Key_Arr(.i.) := ' ';
         Key_Pt@.First_Name_Arr(.i.) := ' ';
      end;

    Source_Target_Flag := false;
    Target_Flag := false;
    Last_Name_Flag := true;
    Char_No_First := 1;
end;
```

```
{*********************************************************
 * Function More_Chars_Left_For_Key checks if there is
 * more input chars and sends boolean value back.
 *********************************************************}

function More_Chars_Left_For_Key : boolean;

begin

    if (((Char_No > NUM_IDENT_CHAR) or
       (Char_No_First > NUM_IDENT_CHAR))
       or eoln(Names)) or eof(Names) then
      begin
         More_Chars_Left_For_Key := false;
         readln(Names);
      end
    else
         More_Chars_Left_For_Key := true;
end;



{*********************************************************
 * Procedure Read_A_Char reads a name and stores it
 * based on the flag and character position variables.
 *********************************************************}

procedure Read_A_Char;
begin
    read(Names, Key_Char);

    if Last_Name_Flag then
      begin
         Key_Pt@.Key_Arr(.Char_No.) := Key_Char;
         Char_No := Char_No + 1;
      end
    else
      begin
         Key_Pt@.First_Name_Arr(.Char_No_First.)
                                              := Key_Char;
         Char_No_First := Char_No_First + 1;
      end;


  if (Key_Char = ' ') and (not Source_Target_Flag) then
      begin
         read(Names, Key_Char);
         if ord(Key_Char) > ord(DISCRIMINATOR) then
           begin
              Target_Flag := true;
```

```
          end;
        Key_Pt@.First_Name_Arr(.Char_No_First.)
                                              := Key_Char;
        Char_No_First := Char_No_First + 1;
        Source_Target_Flag := true;
        Last_Name_Flag := false;
      end;
end;


{*********************************************************
 * Procedure Attach_Key_To_Source_Or_Target_Relations
 * attaches the key record to either the target list
 * or the source list based on the flag.
 *********************************************************}

procedure Attach_Key_To_Source_Or_Target_Relations;

begin
    if Target_Flag then
      begin
        Key_Pt@.next := Hd_Target_Pt;
        Hd_Target_Pt := Key_Pt;
        Target_Count := Target_Count + 1;
      end
    else
      begin
        Key_Pt@.next := Hd_Source_Pt;
        Hd_Source_Pt := Key_Pt;
        Source_Count := Source_Count + 1;
      end;
end;


{--------- Form_Source_And_Target_Relations -----------}

begin
    for Key_No := 1 to MAX_NUM_KEYS do
      begin
        Init_While_Do_Loop;
        While More_Chars_Left_For_Key do
          begin
            Read_A_Char;
          end;
        Attach_Key_To_Source_Or_Target_Relation;
      end;


if SIM_DEBUG then
 begin
```

```
writeln(out);
write(out,' If the first character of a first name is');
write(out,' between "A" ');
writeln(out,'and "', DISCRIMINATOR,'",');
write(out,' the name will be included');
writeln(out,' in the source relation.');
write(out,' Otherwise, the name will be included');
writeln(out,' in the target relation.');
writeln(out);
writeln(out);
write(out,' ------------SOURCE-------------------------');
writeln(out,'-----TARGET-----------------');
writeln(out);
pt1 := Hd_Source_Pt;
pt2 := Hd_Target_Pt;
k1 := 1;
while ((pt1 <> nil) or (pt2 <> nil)) and
                    (k1 <= MAX_NUM_KEYS) do
  begin
     write(out,' ');
     if pt1 <> nil then
       begin
          for k2 := 1 to NUM_IDENT_CHAR do
               write(out,pt1@.Key_Arr(.k2.));
          for k2 := 1 to NUM_IDENT_CHAR do
               write(out,pt1@.First_Name_Arr(.k2.));
          if pt1@.next <> nil then
               pt1 := pt1@.next
          else
               pt1 := nil;
       end
     else
       begin
          for k2 := 1 to NUM_IDENT_CHAR*2 do
               write(out,' ');
       end;

     write(out,'          ');
     if pt2 <> nil then
       begin
          for k2 := 1 to NUM_IDENT_CHAR do
               write(out,pt2@.Key_Arr(.k2.));
          for k2 := 1 to NUM_IDENT_CHAR do
               write(out,pt2@.First_Name_Arr(.k2.));
          if pt2@.next <> nil then
              pt2 := pt2@.next
          else
              pt2 := nil;
       end;

     writeln(out);
```

```
      kl := Kl + 1;

   end;
end;
end;


{*********************************************************
 * Procedure Hash_Source_And_Target_Relations calls
 * Hash_Relation procedure to hash tuples in the relation.
 * *******************************************************}

procedure Hash_Source_And_Target_Relations;

var
    point : LINK;
    address : integer;
    Key_Char : char;
    index : integer;
    i, j : integer;
    Source_Flag : boolean;
    Addr_Array : Addr_Type_Array;
    ok : boolean;
    ptl : LINK;


{*********************************************************
 * Procedure Hash_Relation hashes source relation first,
 * marking corresponding bit array store. Then it hashes
 * tuples in the target relation examining if each tuple
 * is necessary or not. It the tuple is unnecessary,
 * it is discarded immediately. The filtering process is
 * also applied to the source tuples by calling
 * procedure Eliminate_Needless_Source_Tuples.
 *******************************************************}

procedure Hash_Relation (pt : LINK; Source_Relation :
                                        boolean);

var
    Char_No, Bit_No : integer;
    Key_Pt : LINK;
    Next_Pt : LINK;
    i : integer;
    Coder_No : integer;
```

```
{*********************************************************
 * Procedure Look_Up_Char_In_Prime_Num_Table finds out
 * index number to the prime table which contains the
 * character identical to the input character.
 *********************************************************}

procedure Look_Up_Char_In_Prime_Num_Table(var idx:integer);

var  found : boolean;
     j : integer;

begin
    j := 1;
    found := false;
    repeat
     if ASCII_Stack(.Stk_Pt.).ASCII_Arr(.Char_No,j.).ch
                                          = Key_Char then
         begin
              found := true;
              idx := j;
         end
      else
         begin
              j := j + 1;
         end;
    until found or (j > NUM_ASCII_CHAR);

     if (j > NUM_ASCII_CHAR) and (not found) then
          idx := 64;

end;


{*********************************************************
 * Procedure Save_Binary_Prime_Num looks up the
 * corresponding prime number in the ROM table, and
 * copies the number in binary form to the key array
 * (or key register).
 *********************************************************}

procedure Save_Binary_Prime_Num(idx: integer);

var  i, j : integer;

begin

for i := Stk_Pt to MAX_STACK_SIZE do
 begin
    with Bool_Stack(.i.).Bool_Key_Arr(.Char_No.) do
    begin
```

```
    for j := 1 to MAX_BOOL_DIGIT do
      begin
         with ASCII_Stack(.i.) do
              bool_prime(.j.) :=
               ASCII_Arr(.Char_No,idx.).bool_prime(.j.);
      end;

    end;  {with}
 end;

end;


{*********************************************************
 * Boolean function EX_OR receives two boolean input
 * variables and exclusive-ORs the two inputs to send
 * the resulting boolean output to the calling program.
 *********************************************************}

function EX_OR (Bit_X, Bit_Y: boolean): boolean;

begin

    if Bit_X and Bit_Y then EX_OR := false

    else if Bit_X and (not Bit_Y) then EX_OR := true

    else if (not Bit_X) and Bit_Y then EX_OR := true

    else if (not Bit_X) and (not Bit_Y) then EX_OR
                                              := false;

end;


{*********************************************************
 * Procedure First_Level_Ex_Oring receives a hash coder
 * number as an integer input to select a right element
 * in the stack. Then this procedure exclusive-ORs the
 * two input bits or prime numbers and stores the result
 * into proper spot in the temporary array. Therefore,
 * this procedure simulates  the first (or highest)
 * level of the downward complete binary tree of
 * exclusive-OR module.
 *********************************************************}

procedure First_Level_Ex_Oring(Code_No : integer);
```

```
var  i, j : integer;

begin

    i := 1;
    j := 1;

    repeat
       with Bool_Stack(.Code_No.) do
         EXOR1_Arr(.j.) :=
             EX_OR(Bool_Key_Arr(.i.).bool_prime(.Bit_No.)
             ,Bool_Key_Arr(.i+1.).bool_prime(.Bit_No.));
         i := i + 2;
         j := j + 1;
    until (j > NUM_FIRST_EXOR);

end;



{*********************************************************
 * Procedure Second_Level_Ex-Oring exclusive-ORs again
 * the resulting bits calculated by the procedure
 * First_Level_Ex_Oring, and stores the results into
 * temporary array for next exclusive-OR operation.
 *********************************************************}

procedure Second_Level_Ex_Oring;

var  i, j : integer;

begin

    i := 1;
    j := 1;

    repeat
         EXOR2_Arr(.j.) := EX_OR(EXOR1_Arr(.i.),
                                 EXOR1_Arr(.i+1.));
         i := i + 2;
         j := j + 1;
    until (j > NUM_SECOND_EXOR);

end;



{*********************************************************
 * Procedure Third_Level_Ex_Oring exclusive-ORs again
 * the resulting bits calculated by the procedure
 * Second_Level_Ex_Oring, and stores the results into a
```

```
 * temporary array for next level Exclusive_OR operation.
 ******************************************************}

procedure Third_Level_Ex_Oring;
var  i, j : integer;

begin

    i := 1;
    j := 1;

    repeat
         EXOR3_Arr(.j.) := EX_OR(EXOR2_Arr(.i.),
                                 EXOR2_Arr(.i+1.));

         i := i + 2;
         j := j + 1;
    until (j > NUM_THIRD_EXOR);

end;



{******************************************************
 * Procedure Last_Ex_Oring_And_Store_An_Addr_Bit
 * exclusive_ORs the two input bits produced by the
 * procedure Third_Level_Ex_Oring, and produces a
 * final resulting bit of a hash address.
 ******************************************************}

procedure Last_Ex_Oring_And_Store_An_Addr_Bit
                          (Code_No : integer);

begin
  with Code_Stack(.Code_No.) do
     Hashed_Addr(.Bit_No.) := EX_OR(EXOR3_Arr(.1.),
                                    EXOR3_Arr(.2.));
end;



{******************************************************
 * Procedure Bool_To_Int_Convert converts a hash
 * address produced in each participating hash coder
 * to an integer number and stores the number into
 * an array of hash addresses. It repeats this process
 * for every participating hash coder.
 ******************************************************}

procedure Bool_To_Int_Convert
                        (var addrs : Addr_Type_Array);
```

```
var  sum : integer;
     i, j : integer;
     offset : integer;

begin
    for i := 1 to MAX_STACK_SIZE do
      begin
         addrs(.i.) := NUL;
      end;

    for i := Stk_Pt to MAX_STACK_SIZE do
      begin
         sum := 0;
         for j := MAX_BUCKET_ADDR_BITS+Stk_Pt
                                    downto 1+Stk_Pt do
           begin
              with Code_Stack(.i.) do
                begin
                   if Hashed_Addr(.j.) then
                        sum := 2 * sum + 1
                   else
                        sum := 2 * sum;
                end;
           end;
         addrs(.i.) := sum;
      end;
end;



{-------------- Hash_Relation ------------------------}

begin
    while pt <> nil do
      begin
         for Char_No := 1 to NUM_IDENT_CHAR do
           begin
              Key_Char := pt@.Key_Arr(.Char_No.);

           {Read character by character
            for a key looking up the corresponding prime
            number and convert it to binary number}
              Look_Up_Char_In_Prime_Num_Table(index);

              Save_Binary_Prime_Num(index);

           end;
```

```
          for Coder_No := Stk_Pt to MAX_STACK_SIZE do
            begin

          {Get the first digit of the binary prime numbers
           which converted from a character, and
           do the Exclusive-Or operation to get the first
           bit for the hashed address. Repeat this
           process up to the last digit.}

              for Bit_No :=1 to MAX_BOOL_DIGIT do

                begin
                   First_Level_Ex_Oring(Coder_No);
                   Second_Level_Ex_Oring;
                   Third_Level_Ex_Oring;
                   Last_Ex_Oring_And_Store_An_Addr_Bit
                                              (Coder_No);
                end;
            end;

          Bool_To_Int_Convert(Addr_Array);

          Hash_Count := Hash_Count + 1;

if SIM_DEBUG then
  begin
    write(out,' ');
    for i:= 1 to NUM_IDENT_CHAR do
         write(out,pt@.Key_Arr(.i.));


    for i := 1 to MAX_STACK_SIZE do
         write(out,Addr_Array(.i.):7);
    writeln(out);
  end;


    with E_Count(.Stk_Pt.) do
      begin
        Key_Pt := pt;
        Next_Pt := pt@.next;
        if Source_Relation then
          begin
            {Total_Source := Total_Source + 1 }
            with stack(.Stk_Pt.) do
               begin
                  Key_Pt@.next := Source_Bucket
                                   (.Addr_Array(.Stk_Pt.).);
                  Source_Bucket(.Addr_Array(.Stk_Pt.).)
                          := Key_Pt;
               end;
```

```
                    for i := Stk_Pt to MAX_STACK_SIZE do
                      stack(.i.).Bit_Arr(.Addr_Array(.i.).)
                              := true;
            end
          else
            begin
               ok := true;
               i := Stk_Pt;
               Total_Target := Total_Target + 1;
               while ok and (i <= MAX_STACK_SIZE) do
                 begin
                    with stack(.i.) do
                      begin
                       if not Bit_Arr(.Addr_Array(.i.).)
                       then
                             ok := false;
                      end;
                    i := i + 1;
                 end;

               if ok then
                 begin
                    with stack(.Stk_Pt.) do begin
                    Key_Pt@.next :=
                    Target_Bucket(.Addr_Array(.Stk_Pt.).);
                    Target_Bucket(.Addr_Array(.Stk_Pt.).)
                                        := Key_Pt;
                    end;
                 end
               else
                 begin
                    D_Target := D_Target + 1;
                 end;
            end;
          pt := Next_Pt;
        end; {with}
    end; {while}
end;


{*********************************************************
 * Procedure Eliminate_Needless_Source_Tuples
 * eliminates unnecessary source tuples examining the
 * buckets and sets the corresponding bit in the bit
 * array store to false(0).
 *********************************************************}

procedure Eliminate_Needless_Source_Tuples;
```

```
var
    i : integer;
    pt : LINK;
    S_Count : integer;

begin
    for i := 0 to BUCKET_SIZE do
      begin
         with stack(.Stk_Pt.) do
           begin
              S_Count := 0;
              pt := Source_Bucket(.i.);
              while pt <> nil do
                begin
                   with E_Count(.Stk_Pt.) do
                     begin
                        Total_Source := Total_Source + 1;
                        S_Count := S_Count + 1;
                     end;
                   pt := pt@.next;
                end;
              if Target_Bucket(.i.) = nil then
                begin
                   with E_Count(.Stk_Pt.) do
                     begin
                        D_Source := D_Source + S_Count;
                     end;
                   Source_Bucket(.i.) := nil;
                   Bit_Arr(.i.) := false;
                end; {if}
           end; {with}
      end;  {for}
end; {Eliminate_Needless_Source_Tuples}



{--- Hash_Source_And_Target_Relations start from here ---}

begin   {Hash_Source_And_Target_Relations}
    Source_Flag := true;
    point := Hd_Source_Pt;
    Hash_Relation (point, Source_Flag);


if SIM_DEBUG then
 begin
    for j := Stk_Pt to MAX_STACK_SIZE do
      begin
         writeln(out);
         write(out,' ------------');
```

```
            writeln(out,' BIT_ARRAY at Stack Level : ',j:1);
            write(out,'------------ ');
            for i := 0 to BUCKET_SIZE do
              begin

                 if stack(.j.).Bit_Arr(.i.) then
                      write(out,'1')
                 else
                      write(out,'0');

                 if (i mod 50) = 49 then
                   begin;
                      writeln(out);
                   end;
                 if (i mod 10) = 9 then
                      write(out,' ');
              end;
            writeln(out);
         end;
         writeln(out);

 end;


      Source_Flag := false;
      point := Hd_Target_Pt;
      Hash_Relation (point, Source_Flag);

      Eliminate_Needless_Source_Tuples;



if SIM_DEBUG then
 begin
 writeln(out);
 write(out,' Hash_Source_And_Target called ');
 writeln(out,'at Stack Level : ',Stk_Pt :1);
 with stack(.Stk_Pt.) do
      begin
         for i := 0 to BUCKET_SIZE do
           begin

              ptl := Source_Bucket(.i.);
              if ptl <> nil then
                begin
                   writeln(out);
                   write(out,' --- Source Bucket No. ');
                   writeln(out,i:1,' ---');
                end;
              while ptl <> nil do
                begin
```

```
                    write(out,' ');
                    for j := 1 to NUM_IDENT_CHAR do
                        write(out,ptl@.Key_Arr(.j.));

                    for j := 1 to NUM_IDENT_CHAR do
                        write(out,ptl@.First_Name_Arr(.j.));
                    writeln(out);
                    ptl := ptl@.next;
                end;

              ptl := Target_Bucket(.i.);
              if ptl <> nil then
                begin
                    writeln(out);
                    write(out,' --- Target Bucket No. ');
                    writeln(out,i:3,' ---');
                end;
              while ptl <> nil do
                begin
                    write(out,' ');
                    for j := 1 to NUM_IDENT_CHAR do
                        write(out,ptl@.Key_Arr(.j.));

                    for j := 1 to NUM_IDENT_CHAR do
                        write(out,ptl@.First_Name_Arr(.j.));
                    writeln(out);
                    ptl := ptl@.next;
                end;
          end;  {for}
      end;  {with}
 end;
end;



{*********************************************************
 * Procedure Clear_Current_Upper_Part_Of_Stack clears
 * current and upper part of bit array stores and
 * assigns null value to all buckets.
 *********************************************************}

procedure Clear_Current_Upper_Part_Of_Stack;

var
    i, j : integer;
begin
    for i := Stk_Pt to MAX_STACK_SIZE do
      begin
        with stack(.i.) do
          begin
            for j := 0 to BUCKET_SIZE do
```

```
                begin
                   Bit_Arr(.j.) := false;
                   Source_Bucket(.j.) := nil;
                   Target_Bucket(.j.) := nil;
                end;
             Next_Bit := NUL;
          end;
      end;
end;


{*************************************************************
 * Function Only_One_Bit_Set_After_Hash simulates hash
 * address comparators to provide a condition code for
 * checking if only one kind of hash address has been
 * produced by examining all the bits in the participating
 * bit array store(s). If only one bit is set in each
 * participating BASs, the condition code gets true
 * value; otherwise, it gets false value.
 *************************************************************}

function Only_One_Bit_Set_After_Hash(var addr: integer) :
                                                    boolean;

var
    flag : boolean;
    done : boolean;
    i, j : integer;

begin
    i := Stk_Pt;
    addr := NUL;
    done := false;
    while (not done) and (i <= MAX_STACK_SIZE) do
      begin
         flag := false;
         with stack(.i.) do
           begin
              j := 0;
              while (not done) and (j <= BUCKET_SIZE) do
                begin
                   if Bit_Arr(.j.) and (not flag) then
                     begin
                        flag := true;
                        if i = Stk_Pt then
                          begin
                             addr := j;
                             Addr_Num := j;
                          end;
                     end
```

```
                    else
                         if Bit_Arr(.j.) and flag then
                           begin
                              done := true;
                              if i = Stk_Pt then
                                   Next_Bit := j;
                           end;
                    j := j + 1;
               end;
          end;

        i := i + 1;
      end;
    if done then
      begin
        Only_One_Bit_Set_After_Hash := false;
      end
    else
      begin
        Only_One_Bit_Set_After_Hash := true;
        if addr = NUL then
             addr := EMPTY;
      end;
end;


{*********************************************************
 * Procedure Merge_Relations_And_Print_Out simulates the
 * merge process of join executed by the host processor
 * without having key comparisons for the final screening
 * since nearly all unnecessary data have already been
 * filtered.
 *********************************************************}

procedure Merge_Relations_And_Print_Out (addr: integer);

var
    Pt_Source, Pt_Target, Pt_T : LINK;
    i : integer;

begin
 if addr <> EMPTY then
  begin
    if SIM_DEBUG then
      begin
      write(out,' Merge Relations At Stack Level ');
      write(out,Stk_Pt : 1);
      writeln(out,' with  Bucket Number ',addr : 3);
      end;
    if OUTPUT_FLAG then writeln(out);
```

```
    with stack(.Stk_Pt.) do
      begin
         Pt_Source := Source_Bucket(.addr.);
         Pt_Target := Target_Bucket(.addr.);
      end;
    while Pt_Source <> nil do
      begin
         Pt_T := Pt_Target;
         while Pt_T <> nil do
           begin
              If OUTPUT_FLAG then
                begin
                   write(out,' ');
                   for i := 1 to NUM_IDENT_CHAR do
                    write(out,Pt_Source@.Key_Arr(.i.));

                   for i := 1 to NUM_IDENT_CHAR do
                    write(out,
                          Pt_Source@.First_Name_Arr(.i.));

                   for i := 1 to NUM_IDENT_CHAR do
                    write(out,Pt_T@.Key_Arr(.i.));

                   for i := 1 to NUM_IDENT_CHAR do
                    write(out,Pt_T@.First_Name_Arr(.i.));
                end;
              Pt_T := Pt_T@.next;
              Result_Count := Result_Count + 1;
              if OUTPUT_FLAG then writeln(out);
           end;
         if OUTPUT_FLAG then
           begin
              writeln(out);
              writeln(out);
           end;
         Pt_Source := Pt_Source@.next;
      end;
   end
 else
   begin
    if SIM_DEBUG then
         writeln(out,' There are no keys to be merged.');
   end;
end;


{*********************************************************
 * Procedure Save_Next_Bucket_Addr simulates next bit
 * address register in each hash address comparator.
 * Initial value in this register is gained by checking
```

```
 * each individual bits in the current bit array store.
 ******************************************************}

procedure Save_Next_Bucket_Addr;

var
    i : integer;
    found : boolean;
    num : integer;

begin
    num := NUL;
    found := false;
    i := Addr_Num + 1;
    while not found and (i <= BUCKET_SIZE) do
      begin
         if stack(.Stk_Pt.).Bit_Arr(.i.) then
           begin
              stack(.Stk_Pt.).Next_Bit := i;
              found := true;
           end;
         i := i + 1;
      end;

    if (not found) and (i > BUCKET_SIZE) then
         stack(.Stk_Pt.).Next_Bit := NUL;
end;



{******************************************************
 * Boolean function No_More_Next_Bucket_Addr checks whether
 * the next bit address register has null value or not
 * If so, it sends the true value.
 ******************************************************}

function No_More_Next_Bucket_Addr: boolean;

begin
    if stack(.Stk_Pt.).Next_Bit = NUL then
      begin
         Addr_Num := NUL;
         No_More_Next_Bucket_Addr := true;
      end
    else
      begin
         No_More_Next_Bucket_Addr := false;
         Addr_Num := stack(.Stk_Pt.).Next_Bit;
      end;
end;
```

```
{*********************************************************
 * Procedure Assign_Source_And_Target assigns the
 * header pointers of both source and target lists to
 * the temporary header variables based on the saved
 * next bit address in the next bit address register.
 *********************************************************}

procedure Assign_Source_And_Target;

begin
 if SIM_DEBUG then
 begin
    writeln(out);
    write(out,' Assign_Source_And_T ==>');
    write(out,' Address Number : ');
    write(out,Addr_Num : 3);
    writeln(out,'  Stack Number : ',Stk_Pt :1);
 end;
    Hd_Source_Pt :=
              stack(.Stk_Pt.).Source_Bucket(.Addr_Num.);
    Hd_Target_Pt :=
              stack(.Stk_Pt.).Target_Bucket(.Addr_Num.);
end;



{*********************************************************
 * Procedure pop deletes one bit array store from the
 * top of the stack by lowering the stack pointer by one.
 *********************************************************}

procedure pop;

begin
    Stk_Pt := Stk_Pt - 1;
    if Stk_Pt < 1 then
         writeln(out,' Stack Underflow !');

if SIM_DEBUG then
  begin
    writeln(out);
    write(out,' Popped, Stack Pointer is ',Stk_Pt:1);
    writeln(out,' from ',Stk_Pt+1:1);
  end;
end;
```

```
{*********************************************************
 * Procedure push moves up the stack pointer so that
 * a bit array store is inserted onto the top of the
 * stack.
 *********************************************************}

procedure push;

begin
    Stk_Pt := Stk_Pt + 1;
    if Stk_Pt > MAX_STACK_SIZE then
         writeln(out,' Stack Overflow !');

    if SIM_DEBUG then
      begin
         writeln(out);
         write(out,' Pushed, Stack Pointer is ',Stk_Pt:1);
         writeln(out,' from ',Stk_Pt-1:1);
      end;
end;



{*********************************************************
 * Function Bottom_Of_Stack tells if the stack pointer
 * is pointing to the bottom element of the stack.
 *********************************************************}

function Bottom_Of_Stack : boolean;

begin
    if Stk_Pt = 1 then
         Bottom_Of_Stack := true
    else
         Bottom_Of_Stack := false;
end;



{*********************************************************
 * Procedure Print_Statistics prints out the statistical
 * results of the simulation of the new join operation.
 *********************************************************}

procedure Print_Statistics;

var
    i : integer;

begin
 writeln(out);
```

```
writeln(out);
writeln(out);
write(out,' If the first character of a first name is ');
writeln(out,'between "A" and "', DISCRIMINATOR,'",');
writeln(out,' the name will be included');
writeln(out,' in the source relation.');
write(out,' Otherwise, the name will be included');
writeln(out,' in the target relation.');
writeln(out);
writeln(out);
writeln(out);
write(out,' STACK LEVEL  TOTAL TUPLES   DISCARDED ');
writeln(out,'TUPLES  FILTERED RATIO   OVERALL RATIO ');
write(out,'_____________________________________________');
writeln(out,'_____________________________________');
writeln(out);

for i := MAX_STACK_SIZE downto 1 do begin
  with E_Count(.i.) do begin

    write(out,'  ',i:3,'          ',Total_Source:4);
    write(out,'             ',D_Source:4);
    write('               ');

    if Total_Target <> 0 then
       begin
         write(out,(D_Source/Total_Source)*100:7:2);
         write(out,'%        ');
       end
    else
         write(out,'    NONE        ');

    if Target_Count <> 0 then
       write(out,(D_Source/Source_Count)*100:7:2,'%');
       writeln(out,'  <------- SOURCE RELATION ');

       write(out,'               ',Total_Target:4);
       write(out,'             ',D_Target:4);
       write(out,'               ');

       if Total_Target <> 0 then begin
          write(out,(D_Target/Total_Target)*100:7:2);
          write(out,'%        ');
          end
       else
          write(out,'    NONE        ');

       if Target_Count <> 0 then
          write(out,(D_Target/Target_Count)*100:7:2,'%');
       writeln(out,'  <------- TARGET RELATION ');
       writeln(out);
```

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




{************ MAIN PROGRAM STARTS HERE *****************}

begin

Initialization;

Form_Source_And_Target_Relations;

repeat
  Clear_Current_Upper_Part_Of_Stack;

  Hash_Source_And_Target_Relations;

  if Only_One_Bit_Set_After_Hash(Hash_Value) then
  begin
     Merge_Relations_And_Print_Out(Hash_Value);
     if No_More_Next_Bucket_Addr then
     begin
```

```
if Bottom_Of_Stack then
   finish := true
else
begin
   pop;
   if No_More_Next_Bucket_Addr then
   begin
      if Bottom_Of_Stack then
         finish := true
      else
      begin
         pop;
         if No_More_Next_Bucket_Addr then
         begin
            if Bottom_Of_Stack then
               finish := true
            else
            begin
               pop;
               if No_More_Next_Bucket_Addr then
               begin
                  if Bottom_Of_Stack then
                     finish := true
                  else
                  begin
                     pop;
                     if No_More_Next_Bucket_Addr
                     then begin
                        if Bottom_Of_Stack then
                           finish := true
                        else
                        begin
                           Assign_Source_And_Target;
                           Save_Next_Bucket_Addr;
                           push;
                        end;
                     end;
                  end;
               end
               else
               begin
                  Assign_Source_And_Target;
                  Save_Next_Bucket_Addr;
                  push;
               end;
            end;
         end
         else
         begin
            Assign_Source_And_Target;
            Save_Next_Bucket_Addr;
```

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

   Print_Statistics;

   end.
```

# BIBLIOGRAPHY


<AUER1> Auer, H., et al. "RDBM-A Relational Database Machine." Information Systems, Vol. 6, No. 2, 1981: 91-100.

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

<BABB2> Babb, E. "Joined Normal Form: A Storage Encoding for Relational Databases." ACM Transactions on Database Systems, Vol. 4, No.1, Dec. 1982: 588-614.

<BANC1> Bancilhon, F., and Scholl, M. "Design of a Backend Processor for a Data Base Machine." Proceedings of ACM's SIGMOD 1980 International Conference on Management of Data, May 1980: 93-93g.

<BECK1> Beck, M., et al. "Sorting Large Files on a Backend Multiprocessor." IEEE Transactions on Computers, Vol. 37, No. 7, Jul. 1988: 769-78.

<BERN1> Bernstein, P. A., and Chiu, D. W. "Using Semi-Joins to Solve Relational Queries." JACM, Vol. 28, No. 1, Jan. 1981: 25-40.

<BITT1> Bitton, D., et al. "Parallel Algorithms for the Execution of Relational Database Operations." ACM Transactions on Database Systems, Vol. 8, No. 3, Sep. 1983: 324-53.

<BLAS1> Blasgen, M. W., and Eswaran, K. P. "Storage and Access in Relational Data Bases." IBM System Journal, Vol. 16, No. 4, 1977: 363-77.

<BORA1> Boral, H., and Dewitt, D. "Processor Allocation Strategies for Multiprocessor Database Machines." ACM Transactions on Database Systems, Vol. 6, No. 2, Jun. 1981: 227-54.

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

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

<CART1> Carta, D. G. "Two Fast Implementations of the 'Minimal Standard' Random Number Generator." CACM, Vol. 33, No. 1, Jan. 1990: 87-8.

<CAVA1> Cavanagh, J. J. F. Digital Computer Arithmetic Design and Implementation. McGraw-Hill, 1984.

<CODD1> Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." CACM, Vol. 13, No. 6, Jun. 1970: 377-87.

<DATE1> Date, C. J. An Introduction to Database Systems. Reading: Addison-Wesley, 1986.

<DEFI1> DeFiore, C. R., and Berra, P. B. "A Quantitative Analysis of the Utilization of Associative Memories in Data Management." IEEE Transactions on Computers, Vol. C-23, No. 2, Feb. 1974: 121-33.

<DEWI1> DeWitt, D. J. "DIRECT-A Multiprocessor Organization for Supporting Relational Database Management System." IEEE Transactions on Computers, Vol. C-28,

Jun. 1979: 395-406.

<DEWI2> DeWitt, D. J. "Query Execution in DIRECT." Proceedings of the ACM SIGMOD 1979 International Conference on Management of Data, Boston, Mass. , May 1979: 13-22.

<DEWI3> DeWitt, D. J., and Gerber, R. "Multiprocessor Hash-Based Join Algorithms." Proceedings of the Eleventh International Conference on Very Large Data Bases, Stockholm , 1985: 151-64.

<DEWI4> DeWitt D. J., et al. "A Performance Analysis of the Gamma Database Machine." Proceedings of the 1988 SIGMOD Conference, Jun. 1988: 350-60.

<ENBO1> Enbody, R. J., and Du, H. C. "Dynamic Hashing Schemes." ACM Computing Surveys, Vol. 20, No. 2, Jun. 1988: 85-113.

<GOOD1> Goodman, J. R., and Sequin, C. H. "Hypertree: A Multiprocessor Interconnection Topology." IEEE Transactions on Computers, Vol. C-30, No. 12, 1981: 923-33.

<GROG1> Grogono, P. Programming in Pascal. Addison Wesley, 1980.

<HANA1> Hanan, M., and Palermo, F. P. "A Application of Coding Theory to a File Address Problem." IBM Journal R & D, Vol. 7, No. 2, Apr. 1963: 127-9.

<HSIA1> Hsiao, D. K. Advanced Database Machine Architecture. Englewood Cliffs: Prentice Hall, 1983.

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

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

<KNUT2> Knuth, D. E. The Art of Computer Programming. Vol. 2, Seminumerical Algorithms, Reading: Addison-Wesley, 1975.

<LEE1> LEE, C. Y., and PAUL, M. C. "A Content Addressable Distributed Logic Memory with Applications to Information Retrieval." Proceedings of the IEEE, Vol. 51, No. 6, Jun. 1963: 924-32.

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

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

<MART1> Martin, J. Computer Data-Base Organization. Englewood Cliffs: Prentice-Hall, Inc., 1977.

<MARY1> Maryanski, F. J. "Backend Database Systems." ACM's Computing Surveys, Vol. 12, No. 1, Mar. 1980: 3-25.

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

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

<MELL1> Mellen, G. E. "Cryptology, computers, and common sense." Computers and Security. Edited by C. T. Dinardo, Montvale: AFIPS Press, 1978.

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

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

<MOTO2> Motorola, Inc. MC68881/MC68882 Floating-Point Coprocessor User's Manual. Englewood Cliffs: Prentice Hall, 1987.

<OZKA1> Ozkarahan, E. Database Machines and Database Management. Englewood Cliffs: Prentice Hall, 1986.

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

<QADA1> Qadah, G. Z., and Irani, K. B. "The Join Algorithms on a Shared-Memory Multiprocessor Database Machine." IEEE Transactions on Software Engineering, Vol. 14, No. 11, Nov. 1988: 1668-83.

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

<RICH1> Richardson, J. P., et al. "Design and Evaluation of Parallel Pipelined Join Algorithms." ACM SIGMOD, Vol. 16, No. 3, Dec. 1987: 399-409.

<SCHA1> Schay, G., and Raver, N. "A Method for Key-to-Address Transformation." IBM Journal R & D, Vol. 7, No. 2, 1963: 121-6.

<SCHN1> Schneider, D. A., and DeWitt, D. J. "A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment." Proceedings of the 1989 ACM SIGMOD, Vol. 18, No. 2, Jun. 1989: 110-21.

<SHAP1> Shapiro, L. D. "Join Processing in Database Systems with Large Main Memories." ACM Transactions on Database Systems, Vol. 11, No.3, Sep. 1986: 239-64.

<SHUL1> Shultz, R. K. "Response Time Analysis of Multiprocessor Computers for Database Support." ACM Transactions on Database Systems, Vol. 9, No. 1, Mar. 1984: 100-132.

<SIEG1> Siegenthaler, T. "Cryptanalysts Representation of Nonlinearly Filtered ML-Sequences." Advances in Cryptology-EUROCRYPT'85, Proceedings of a Workshop on the Theory and Application of Cryptographic Techniques, Linz, Austria, Apr. 1985.

<SMIT1> Smith, D. C., and Smith, J. M. "Relational Database Machines." IEEE Computer, Vol. 12, No. 3, Mar. 1979: 18-38.

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

<STON1> Stonebraker, M. R., et al. "The Design and Implementation of INGRES." ACM Transactions on Database Systems, Vol. 1, No. 3, Sep. 1976: 189-222.

<SU1> Su, S. Y. W. Database Computers Principles, Architectures, and Techniques. New York: McGraw-Hill, 1988.

<TENE1> Tenenbaum, A. M., and Augenstein, M. J. Data Structures Using Pascal. Englewood Cliffs: Prentice Hall, 1986.

<ULLM1> Ullman, J. D. Principles of Database Systems. Rockville: Computer Science Press, 1982.

<VALD1> Valduriez, P., and Gardarin, G. "Join and Semijoin Algorithms for a Multiprocessor Database Machine." ACM Transactions on Database Systems, Vol. 9, No. 1, Mar. 1984: 133-61.

<VALD2> Valduriez, P. "Semi-Join Algorithms for Multiprocessor Systems." Proceedings of ACM's SIGMOD 1982 International Conference on Management of Data, Jul. 1982: 225-33.

<VALD3> Valduriez, P. "Join Indices." ACM Transactions on Database Systems, Vol. 12, No. 2, Jun. 1987: 218-46.

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