Nov. 14, 1995. V2

Implementation of Shin's Algorithm for the Relational Join

Dong-Keun Shin and Arnold Charles Meltzer

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

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

Abstract

Showing a hardware implementation of the Shin's join algorithm, the paper illustrates how the join algorithm efficiently filters out all the unnecessary tuples. In contrast to other join algorithms, the Shin's join algorithm allows join attribute comparisons after nearly one hundred percent of the unnecessary tuples are eliminated. Using only fixed-size hash tables and possessing an inherent characteristic of parallel processing, the Shin's join algorithm is highly recommendable above all other join algorithms.

The Shin's join algorithm can be executed in various ways of parallel processing. The algorithm is divided into two major processes: filtering and merging. The major processes can be executed in parallel. Then, each major process can be further divided into subprocesses, which, too, can be executed in parallel. This algorithm might be implemented in a database computer for an efficient parallel execution of the two major processes. In an initial design phase of a database computer-HIMOD, the join operation is performed by a general purpose processor as a host and a parallel architecture join database coprocessor. The host performs the merging process for the join while the join database coprocessor mainly performs the filtering process for the join. The join database coprocessor is designed to be a speedy filter device which further accelerates the join.

I. INTRODUCTION

Ever since the relational data model was introduced by E. F. Codd's paper [CODD70] in 1970, people have well recognized not only the naturalness of the two dimensional table structure, but also the usefulness of the join relational operation. However, even if they change the database management system in the future, people will still place emphasis on a solution to massive cross-references from an input list to another. Thus, an optimal solution to the join will be considered as the theory of massive cross-referencing. In addition, the join operation is frequently used, and yet, very time-consuming. Therefore, increasing the speed of the join has been a popular issue for more than 20 years.

One of the goals of our research was to provide an optimal algorithmic solution for the

massive cross-references to accelerate the time-consuming join relational database operation. In this paper, Shin's algorithm will be described as a solution for the join. This study will compare the Shin's join algorithm with other join algorithms focusing on the question of their effectiveness of parallel processing. The effectiveness of parallel processing in each join algorithm can be considered as the chief means of improving the speed of the join operation. The paper will also illustrate a filter device which uses the join algorithm to show how the join algorithm can be implemented in hardware.

The join operation concatenates a tuple of the source relation (S) with a tuple of the target relation (T) if the value(s) of the join attribute(s) in this pair of tuples satisfies a pre-specified join condition, and it produces a tuple for the resulting relation (R). The join operation can be illustrated in SQL using a SELECT-FROM-WHERE clause. When X.A is used to indicate that attribute A from relation X is meant and X.* stands for all attributes of relation X, a simple example of the join and the query the example is in Figure 1.

In 1977, Blasgen and Eswaran [BLAS77] described several methods for evaluating a general query, involving project, select, and join relational database operations. They compared the methods based on the number of accesses to secondary storage. In their examination of the join operation, nested-loop and sort-merge algorithms were analyzed and discussed. Both join algorithms will be discussed later in greater detail.

Based on the nested-loop and the sort-merge join methods, tuples will be carried until the last moment, even though they may not be necessary to the resulting relation. Assuming that the number of tuples in the source and target relations is large, but the number of resulting tuples is small, then most of the tuples in the source and target relations may not be needed in producing the output for the join. However, all of those irrelevant tuples are also brought to main memory from secondary storage via the I/O channel. As a consequence the channel becomes congested, which, in turn, creates the aforementioned I/O bottleneck. Many researchers have broached designing a database filter for the join operation to reduce the problem of channel congestion. CAFS [BABB79, SU81], for example, has been designed based on the concept of database filtering.

Since the cost of the main memory has been substantially reduced, the potential of hash join algorithms has received much recognition. DeWitt and Gerber said that the join algorithm based on hashing is more advantageous than nested-loop or sort-merge join algorithms in terms of speed [DEWI85]. However, the hash join algorithms are especially vulnerable to fluctuations in memory availability because they rely heavily on main memory. For each hash join, the requirement for buffer size and subset files is notoriously variable and unreliable. Join algorithms, such as nested-loop, sort-merge, and hash, require frequent join attribute comparisons which may result in more data movements. In 1988, the authors of this paper realized that an optimal algorithmic solution for the join was yet to be found. The Shin's join algorithm is expected to resolve the problems of the currently exiting join algorithms.

In the Section II, the four aforementioned major join methods are illustrated: the nested-loop join algorithm, the sort-merge join algorithm, the hash join algorithm, and the Shin's join algorithm. The time complexities and problems of join algorithms are discussed in Section III. Section IV describes HIMOD - a database computer that may effectively perform the Shin's join algorithm. Finally, in Section V we give concluding remarks and suggest some directions for future research.

II. FOUR MAJOR JOIN ALGORITHMS

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

A. The Nested-Loop Join Algorithm

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

B. The Sort-Merge Join Algorithm

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

C. The Hash Join Algorithm

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

D. The Shin's Join Algorithm

Using a divide and conquer strategy, Shin's join algorithm repeatedly divides the source and target relations by a maximum of five functionally different hash coders and filters out unnecessary tuples whenever possible. After completing a division (or hashing) process, the algorithm checks whether or not the source tuples and the target tuples in a pair of source and target buckets have an identical join attribute. If so, the source and target tuples in the pair of buckets are then merged in order to produce tuples for the resulting relation. Otherwise, the address of the current pair of source and target buckets is saved, and the source and target tuples in the pair of buckets may be further divided by another functionally different hash coder. If a bucket is empty and the corresponding bucket in the pair is not empty, the tuples in the corresponding bucket are not necessary; thus, they are discarded. The algorithm continues

dividing the tuples in a pair of buckets, merging the tuples, or eliminating unnecessary tuples until every tuple in the buckets of created hash tables is either merged or eliminated.

A stack is an essential data structure of the SOFT (Stack Oriented Filter Technique) and the Shin's join algorithm as shown in Figure 2. Each stack item consists of a pair of two hash tables: one for the source tuples and the other for the target tuples. The stack pointer keeps track of the top item of the stack whenever a stack item is pushed into or popped from the top of the stack. (Stacks in Figure 2 depict changes in the stack.) In the process of Shin's join, a maximum of five pairs of hash tables can be created. A source hash table includes 256 bucket pointers for the linked lists of source tuples. A target hash table also includes 256 bucket pointers for the linked lists of target tuples. One may choose proper numbers for the maximum level of the stack and size of the hash table instead of using the stack level and table size mentioned above (i.e., five and 256).

Assuming that both input relations fit in main memory, they are divided into a maximum of 256 linked lists for each relation by the first hash coder, as shown in step 1 in Figure 2. After the source and target tuples are hashed by the first hash coder, the tuples in the source bucket (Si) can match, possibly, with only the tuples in the corresponding target bucket (Ti). If an empty bucket exists, all tuples in the corresponding bucket would be eliminated since they have no potential of being included in the resulting relation. If a DBMS has an insufficient main memory space for input tuples, subset files will be used for the Si and Ti buckets that are not able to reside in main memory.

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

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

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

Shin's Join algorithm provides the termination condition to end further division process as an idea for the SOFT. Whenever the tuples in the pair of source bucket and target bucket are divided by a hash coder, the SOFT checks if the termination condition. If the produced hash addresses in a group of source and target tuples are identical, the termination condition is satisfied. Then the algorithm stops dividing the group and starts merging the source and target tuples.

In parallel architecture filter unit of HIMOD database computer, a maximum of five functionally different hash coders may be involved in checking the termination condition. If their logical ANDed result show that only a single hash address is produced from each involved

hash coder, the group of source and target tuples can be merged without final screening.

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

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

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

In this algorithm, the number of visits to the buckets is proportional to the number of inputs; therefore, traversing the buckets in the hash tables is no cause for concern. In an actual implementation of the algorithm, the bucket traversal may be accelerated if one uses suitable registers to store bit value for each bucket to indicate whether or not the bucket is empty.

As shown in Figure 3, push and pop are the names of the procedures operating in the stack. The procedure push inserts a pair of source and target hash tables onto the top of the stack and increments the stack pointer. The procedure pop deletes a pair of source and target hash tables from the top and decrements the stack pointer. The stack pointer always points to the current pair of hash tables (the item at the top of the stack). By referring to the value in the stack pointer, the function Bottom_Of_Stack can tell whether the stack pointer points to the first (or the lowest) pair of hash tables as the current item of the stack.

In the Shin's join algorithm, there are several other frequently used procedures such as Assign_Source_And_Target, No_More_Next_Bucket_Addr, and Save_Next_Bucket_Addr. The module Assign_Source_And_Target uses the header pointers of both source and target linked lists based on the saved next bucket address of the current pair of hash tables in order that the tuples in the linked lists are processed through the filter again. Each next bucket address is incremented and saved to keep track of the subsequent bucket address. Whenever the procedure Assign_Source_And_Target is called, another next bucket address, which has non-empty buckets for both source and target hash tables, is found and saved by the procedure termed Save_Next_Bucket_Addr. As a result, the procedure push saves the contents of the current pair

of hash tables and increments the stack pointer so that the next upper pair of hash tables may become the current one or the top of the stack.

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

The algorithm shown in Figure 3 is an explanatory version of the main module of a simulation program [SHIN91] for the Shin's join algorithm. The nature of recursion in the algorithm simplifies the architectural structure of the database computer which implements the Shin's join algorithm. The nonrecursive implementation of the algorithm in Figure 3 is useful and practical because the stack has only five items. The nonrecursive algorithm can also be easily implemented in a microprogram.

III. DISCUSSION

When the number of tuples in the source relation (the smaller relation) is S, the number of tuples in the target relation (the larger relation) is T, and the number of tuples in the resulting relation is R, then the time complexity of nested-loop join algorithm is O(S*T). Since the upper bound of S and T is N, O(S*T) can be expressed as O(N*N). The time complexity of the sort-merge algorithm is O((S+T) log (S+T)). It can be represented as O(N log N) since the total number of the input tuples (N) is S+T. Accordingly, in terms of the time complexity, one can assume that the sort-merge join algorithm is superior to the nested-loop join algorithm.

To derive an asymptotic time complexity for 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 the hashing process (B/D) is reduced as the value of D gets larger. Therefore, the time complexity for the hash join algorithm can be simplified as O(S+T+R). Assuming that R is relatively smaller than S+T, it becomes O(S+T). Since S+T is actually the total number of the input tuples(N), the time complexity can be represented as O(N). It is believed that the time complexity of the join operation cannot be better than O(N) since the join attribute in every input tuples must be read at least once.

Considering actual performances, it is hard to rely only on the analysis of the asymptotic time complexity when measuring the speed performance because of the number of accesses to the secondary storage, I/O access time, the condition of interprocess communications, and related overheads. However, in the future, when necessary hardware including main memory is sufficiently provided, the time complexity will be more reliable and will be the most important criteria.

When sufficient main memory is affordable, the hash-based join has the greatest advantage [DEWI85, SCHN89, SHAP86]. The performance of the hash join algorithm is

largely dependent on the distribution performance of a chosen hash function. If the chosen hash function poorly distributes the tuples, the worst case may not be avoidable. Accordingly, the dependence on a chosen hash function is absolute and large in the performance of the hash join algorithm. In contrast to the hash join algorithm, the Shin's join algorithm uses five functionally different hash coders, so it is less dependent on the performance of a chosen hash coder than the hash join algorithm.

The hash join algorithm usually requires a flexible size for the hash table for each hashing process. Such flexibility of a hash table is not recommended for a hardware implementation of hash table in a database machine. The amount of associated buffer space and the number of involved subset files will be so fluctuated that the performance of the hash join algorithm is not as stable as that of the Shin's join algorithm. The fluctuations in main memory consumption is a serious drawback to stabilizing database systems [PANG93]. A complicated bucket tuning process (as a way of resolving this problem) can make the join more complex and cumbersome. The bucket tuning process do not provide the best remedy for the problem of the hash join, because it works only after the whole hash table is built.

Another problem resides in the frequent join attribute comparisons of the hash join algorithm. The architecture of the database computer that implements the join attribute comparison may be complex. Join attribute comparisons will include the comparisons for unnecessary tuples which will not be included in a resulting relation. A join attribute comparison takes a much longer time than the hashing of a join attribute takes. An effective hardware hash coder such as Shin's mapping hash coder [SHIN91] can calculate a hash address within only a few machine cycles. Join attribute comparisons should be performed after all the unnecessary tuples are eliminated.

Because the shin's algorithm requires the fixed-size hash tables and a simple architecture, an intelligent secondary storage device, which frequently accesses disks, may also use the Shin's join algorithm to filter unnecessary tuples effectively. Effective parallel processings, which include parallel read capability in I/O (e.g., disk) devices, are recommended to accelerate the join. The Shin's algorithm can also be used in parallel accesses of secondary storage detect and filter unnecessary tuples efficiently and pass only the necessary tuples to the DBMS.

The Shin's join algorithm requires a maximum of five readings for each join attribute to determine whether the associated tuple is necessary or not. Therefore, the time complexity of the algorithm is represented as O(5*N) and can be simplified as O(N). The time complexity for the traversal through buckets (O(T)) will be O(N) since the number of visits to buckets is proportional to the number of inputs. Therefore, the time complexity is still O(N) since O(N) + O(T) = O(N) + O(N) = O(N).

Comparing the Shin's join algorithm with others, one can see that none of the currently existing join algorithms effectively takes advantage of filtering schemes while the Shin's algorithm filters unwanted data efficiently. The Shin's join algorithm requires only fixed-size hash tables which are favorable for hardware implementation. Moreover, the performance of the Shin's join algorithm has much smaller dependency on the distribution performance of a chosen hash function than that of the hash join algorithm. The simulation for the join was performed by Shin [SHIN91]. The source, target, and correct resulting relations can be presented as proof that the Shin's algorithm logically works.

Before the Shin's algorithm was discovered, people generally thought that among those

which were several well-known hash join algorithms was an optimal solution for the join (i.e., equi-join). Schneider and DeWitt [SCHN89] stated that Hybrid hash-join algorithm is superior except when the join attribute values of the inner relation are non-uniformly distributed and memory is limited. Although the survey of join algorithms is not a simple task because many factors and many uncertainties are involved, the Shin's algorithm should be compared with the hash join algorithms (e.g., the Hybrid) in a fair survey. A great deal of effort will go into this survey not only to increase the speed of the time-consuming join operation but also to approve a theory that provides an optimal solution for massive cross-referencing.

Section IV will describe how the Shin's join algorithm is implemented in a back-end database computer to detect and filter unnecessary data for the join.

IV. DATABASE COMPUTER HIMOD

The parallel computer architecture that facilitates a faster join and Shin's join algorithm, which performs best when using such architecture, are presented in this paper. The database computer which is discussed in this paper is named HIMOD (Highly Modular Relational Database Computer). It uses a back-end join coprocessor fabricated in a single chip [ABD76] in its initial stage of development. The database back-end processor (DBCP) in HIMOD is dedicated to the join database operation. Using the DBCP as a filter device, the HIMOD maximizes the filtering effect in the join process.

This section will describe how Shin's join algorithm can be performed by two major processes such as filtering and merging, and it will show how these processes are executed in parallel. Illustrating architecture of filter unit of the DBCP, it will explicitly explain how the filter unit detects and eliminates unnecessary tuples efficiently.

A. An Overview of HIMOD Architecture

Shin's join method [SHIN91], used in HIMOD, is divided into two processes: filtering and merging. Figure 4 depicts the interface between the host and the back-end, and illustrates the parallel execution of the filtering process and the merging process. As shown in Figure 4, the host requests a join operation of a source relation and a target relation. Then the back-end will receive the request and perform the join of the source and target relations. Filtering unnecessary tuples, the back-end transmits the pointers, which point to the tuple lists, to the host processor whenever it finds the tuples that will be included in the resulting relation. The linked source tuple(s) and target tuple(s) are retrieved from the main memory and merged by the host processor. Therefore, parallelism is exploited in the join operation so that the filtering process and the merging process are concurrently executed by the special purpose back-end and the general purpose processor (host) respectively, as is indicated in Figure 4. The idea behind the Shin's join algorithm in the database computer HIMOD contends that the host processor handles only tuples that are necessary for the resulting relation. The host processor is not burdened with carrying too many join attributes and comparing them for a match. The filtering scheme in HIMOD is accomplished by the Shin's join algorithm.

HIMOD uses a Motorola 68000 family microprocessor as the host (or the front-end as

one might consider) processor. The back-end processor communicates with the host processor through a protocol, which is defined as the M68000 coprocessor interface [MOTO87a]. 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 5. All communications between the HPU and the DBCP are performed with standard M68000 family bus transfers. The DBCP contains a number of coprocessor interface registers, which are addressed by the host processor in the same manner as memory.

B. Architecture of the Host Processor

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

C. Architecture of the Hardware Back-End

The new hardware back-end processor is intended primarily as a database coprocessor(DBCP) to the M68000 family microprocessor unit (HPU). This database coprocessor provides a high performance filter unit which is designed by Shin [SHIN91]. As shown in Figure 6, the database coprocessor is internally divided into three processing elements: the bus interface unit, the coprocessor control unit, and the filter unit. The bus interface unit communicates with the host processor, and the coprocessor control unit sends control signals to the filter unit in order to execute the intended database operation. For both the bus interface unit and the processor control unit, the DBCP uses the conventions of the MC68881 and MC68882 floating-point coprocessor chips [MOTO87b].

1) Bus Interface Unit

The bus interface unit contains the coprocessor interface registers (CIRs), the CIR register select and timing control logic, and the status flags that are used to monitor the status of communications with the host processor. The CIRs are addressed by the host processor in the same manner as memory. All communications between the host processor unit and the DBCP are performed with standard M68000 family bus transfers [MOTO87a]. The M68000 family coprocessor interface is implemented as a protocol of bus cycles during which the host processor reads and writes to these CIRs. An MC68000 family host processor implements this general purpose coprocessor interface protocol in both hardware and microcode.

2) Coprocessor Control Unit

The control unit of the DBCP contains a clock generator, a two-level microcoded sequencer, and a 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 activities. The microsequencer also controls instruction execution and internal processor operations, such as setting condition codes and calculating effective addresses. The microsequencer provides the microinstruction decode logic, the instruction decode register, the instruction decode PLA, and it determines the "next microaddress" generation scheme for sequencing the microprograms. The microcode ROM contains the microinstructions, which specify the steps through which the machine sequences and which control the parallel operation of the functionally equivalent slices of the filter unit.

3) Filter Unit

One of the main tasks of the DBCP is to release the host from tedious database manipulation for the relational join by filtering tuples that do not have any potential for inclusion in the resulting relation. To this end, the DBCP sends only the potential tuples to the host processor. The filter unit of the DBCP is the heart of the coprocessor in determining unnecessary data and discarding them. As shown in Figure 7, the filter unit of the DBCP includes the join attribute extractor, transmittal and retrieval subunit, condition code, stack pointer register, and five functionally different mapping hash coders with associated SAT (source and target single-bit wide memory) and associated HAC (hash address comparator). The join attribute extractor receives a join attribute(s) (or a tuple dependent on given data structure) and transmits the join attribute value to the five attached hash coders. Compared to well-known hash methods (e.g., the division method), the Shin's mapping hash coder distributes keys effectively and the mapping hash coder is much faster and cheaper. Thus the hardware implementation of the mapping hash function [SHIN91] is used in the hash coder of HIMOD. The hash addresses generated by functionally different mapping hash coders using a join attribute as an input key are distinct, but the address calculation times required by functionally different mapping hash coders are always the same and are only a few machine cycles. The property of functional difference in the mapping hash coder is valuable in this application.

The SAT includes two single-bit wide memories (e.g., special registers or RAMs). One memory (source memory) is for a group of source tuples, and another (target memory) is for a group of target tuples. The single-bit wide memory has 256 bits. Each bit in a memory is addressed by a hash address. Each SAT is connected with a hash address register in an associated HAC. The hash address register is equipped with an increment function so that the address register will keep track of the next bucket address to be processed, and will feed it to the connected SAT. Therefore, each SAT has a built-in multiplexer to select the right address at any time as is shown in Figure 8. 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 memories. Therefore, when the tuples in the source relation are scanned, the single-bit wide source memory is marked based on the hash address from the hash coder.

By the same system, when the tuples in the target relation are scanned, the single-bit

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

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

To efficiently determine whether or not the scanned source tuples and target tuples have the same join attribute, a HAC is attached to each of the five hash coders. The HAC is designed so that it sends a signal to the controller to stop dividing the tuples, as is explained below. The HAC consists of a hash address register which keeps a record of the first hash address produced by the corresponding hash coder and the number of exclusive-OR gates, the OR gate, and the JK flip-flop. Each incoming hash address is compared with the first produced hash address, as illustrated in Figure 9.

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

triggers the K input of the JK flip-flop (The output of the JK flip-flop is initially cleared to be '1' by the controller.), so the output of the JK flip-flop becomes '0.' Thus, the five structurally identical HACs in the DBCP generate output signals at the same time.

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

The hardware for hash address comparison, required to detect whether all the join attributes in a file or list are identical, merits elucidation. The purpose of this hardware is to inform the controller whether or not the input file or list should be divided further. If so, the DBCP eventually sends the pointers to the source and target tuples having the same join attribute to the host processor for concatenation. As shown in Figure 10, the five HACs are stacked. Based on the value in the stack pointer register, the 5-to-1 multiplexer selects one from the five inputs. When the stack pointer designates the first (i.e., the lowest) stack level, all the outputs from the HACs are ANDed, and the resulting output of the AND gate (D) is selected by the multiplexer. If the stack pointer specifies the second stack level, the first SAT is saved in the stack and is not written until the controller sends a memory write signal to the SAT. The output of the first HAC is, therefore, excluded from the inputs into the AND gate (C), and outputs of the second, third, fourth, and fifth HACs are ANDed. Likewise, if the indicated stack level is the third level, the first and second SATs are saved in the stack and the multiplexer chooses the output of the AND gate (B), which receives the outputs from the third, fourth, and fifth HACs as inputs. If the indicated stack level is the fifth (the highest) level, the four lower level SATs are saved and the multiplexer selects the output directly from the fifth level HAC. The stack configurations explain the stack in the SOFT as shown in Figure 2.

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 perform a conquer process. In the conquer process, the controller allows the transmittal and retrieval subunit to send pointers to the lists of the necessary source and target tuples to the host processor for a merge. Then the host processor retrieves the source and target tuples using the pointers, and it merges the tuples to produce resulting tuples. The transmittal and retrieval subunit figures out the pointers (addresses in main memory) to the lists of the source and target tuples using the current stack level and a saved bucket address in a selected HAC as inputs. Then the transmittal and retrieval subunit sends the obtained pointers to the wanted tuple lists

to the host for a merge. Even though the output signal indicates that no further division process is necessary, there is fewer than one chance in a trillion (1/(256**5)) that the signal will pass an unwanted key. Although the final screening is not necessary due to the 99.9999999999% filtering effect, the final screening with direct comparisons by the host processor will eliminate the spurious key, if it is present. Because this chance is extremely small, the host processor will not waste time dealing with unnecessary data and the direct comparison of join attributes is not needed.

In addition to the SOFT, the HIMOD database computer may employ the hashed address bit array stores filtering technique in CAFS [BABB79]. The previous design of the HIMOD [SHIN91] uses the filtering technique in CAFS. In order to employ the technique in the HIMOD, a new scheme for the hash coders, so called dynamic hash coders, might be needed due to differences in architectures. In this scheme, each dynamic hash coder generates a distinct hash address based on the current stack level information. At this point, a database computer designer may consider cost versus performance trade-offs. In this paper, the filtering technique in CAFS is not included in the design of the filter unit because authors want to explain the hardware implementation of the Shin's join algorithm clearly.

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

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

V. CONCLUSIONS AND FUTURE RESEARCH

A. Summary

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

Comparing the Shin's join algorithm with others, one can see that none of the currently existing join algorithms effectively takes advantage of any filtering scheme while the new join algorithm filters unwanted data efficiently. Moreover, the Shin's join algorithm has an advantage in a parallel processing because parallelism is one of the important characteristics of

the Shin's join algorithm. In the Shin's join algorithm, filtering and merging processes can be executed in parallel, and the process of filtering tuples in partitioned linked lists can also be divided into subprocesses that are to be executed in parallel.

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

The join database coprocessor repeats the division and filtering process many times in a recursive way; therefore, nearly one hundred percent of the unnecessary tuples are filtered. After repetitive division and filtering processes, the remaining tuples in the source and corresponding target list have an extremely high probability of having identical join attributes. The remaining source and target tuples are sent to the host processor and merged. All other tuples are eliminated before unnecessary comparison of their join attributes begins. This elimination of unnecessary tuples substantially reduces the number of join attribute comparisons. As a result, total data movements in performing a join are radically diminished.

The distinguishable difference between Shin's join algorithm and other hash-based join algorithms is that, in the Shin's join algorithm, the filtering process is combined with the hashing process. Accordingly, unnecessary data are detected and filtered while other join algorithms may carry unwanted tuples up to the last moment of join attribute comparisons.

B. Conclusions

This research has led to a new database computer using the Shin's join algorithm. The algorithm will shorten the time needed for a join, since it frequently filters unnecessary data. On the contrary, the hash join algorithm is not recommended because it requires flexible size for a hash table. Such flexibility of a hash table is not only the cause of sudden and unpredictable main memory consumption but also an obstacle to hardware implementations. Because the Shin's join algorithm requires only fixed size for a hash table, it does not have the problems that hash join algorithms have.

The database computer HIMOD can further accelerate the join because it employs the parallel execution of the filtering process and the merging process to accomplish the Shin's join method. To maximize the speed of the filtering process, the filter unit in the join database coprocessor is organized as a stack. The proposed architecture of the stack oriented filter technique can be employed for a database processor or an intelligent I/O device.

C. Future Research

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

The multiple back-ends database computer would outperform the multiprocessor database computer that uses well known join methods, such as the parallel nested-loop join, the parallel sort-merge join, and the parallel hash join methods. None of the well known methods exploits the filtering mechanism in their parallel join algorithms and none of these methods have an inherent characteristic of parallel processing while the Shin's parallel join algorithm has both advantages. It will be useful to discuss how the Shin's join algorithm will be applied to an intelligent I/O device which accesses disks in parallel and eliminates unnecessary tuples efficiently. A comparative study of these parallel join methods, including Shin's join algorithm, based on the measured response time, may provide a good direction for the increasing of the speed of the join.

ACKNOWLEDGMENTS

We thank Domenico Ferrari, Michael Stonebraker, Arie Segev, Kun-Hee Lee, Steven Schwarz, David Messerschmitt, Diogenes Angelakos, Lotfi Zadeh, Ward Maurer, Michael Feldman, and Helen Shin for their feedbacks and supports. We are thankful to E. Babb, Stanley Su, and David DeWitt for their previous works. Finally, Shin would like to thank Manuel Blum, David Patterson, and Simon Berkovich for their teachings which made him successful in discovering an algorithm for the join.

BIBLIOGRAPHY

[ABD76] Abd-alla, A. M. and Meltzer, A. C. *Principles of Digital Computer Design.* Vol. I, Englewood Cliffs: Prentice Hall, 1976.

[AUER81] Auer, H., et al. "RDBM-A Relational Database Machine." *Information Systems,* Vol. 6, No. 2, 1981: 91-100.

[BABB79] Babb, E. "Implementing a Relational Database by Means of Specialized Hardware." *ACM Transactions on Database Systems,* Vol. 4, No. 1, Mar. 1979:

1-29.

[BABB82] Babb, E. "Joined Normal Form: A Storage Encoding for Relational Databases." *ACM Transactions on Database Systems,* Vol. 4, No. 1. Dec. 1982: 588-614.

[BANC80] 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.

[BITT83] 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.

[BLAS77] Blasgen, M. W. and Eswaran, K. P. "Storage and Access in Relational Data Bases." *IBM System Journal,* Vol. 16, No. 4, 1977: 363-77.

[BORA81] 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.

[CODD70] Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *CACM*, Vol. 13, No. 6, Jun. 1970: 377-87.

[DATE81] Date, C. J. *An Introduction to Database Systems.* Reading: Addison-Wesley, 1981.

[DATE87] Date, C. J. *A Guide to Ingres.* Reading: Addison-Wesley, 1987.

[DEWI85] 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.

[DEWI88] DeWitt D. J., et al. "A Performance Analysis of the Gamma Database Machine." *Proceedings of the 1988 SIGMOD Conference,* Jun. 1988: 350-60.

[GOOD81] Goodman, J. R. and Sequin, C. H. "Hypertree: A Multiprocessor Interconnection Topology." *IEEE Transactions on Computers,* Vol. C-30, No. 12, 1981: 923-33.

[HORO78] Horowitz, E. and Sahni, S. *Fundamentals of Computer Algorithms.* Rockville: Computer Science Press, Inc. 1978.

[HSIA83] Hsiao, D. K. *Advanced Database Machine Architecture.* Englewood Cliffs: Prentice Hall, 1983.

[MARY80] Maryanski, F. J. "Backend Database Systems." *ACM's Computing Surveys*, Vol. 12, No. 1, Mar. 1980: 3-25.

[MOTO87a] Motorola, Inc. *MC68030 Enhanced 32-bit Microprocessor User's Manual.* Motorola, Inc., 1987.

[MOTO87b] Motorola, Inc. *MC68881/MC68882 Floating-Point Coprocessor User's Manual.* Englewood Cliffs: Prentice Hall, 1987.

[PANG93] Pang, H., Carey, M. J., and Miron, L. "Partially Preemptible Hash Joins." *Proceedings of the ACM SIGMOD*, May 1993: 59-68.

[PATT90] Patterson, D. A. and Hennessy, J. L. *Computer Architecture: A Quantitative Approach.* San Francisco: Morgan Kaufmann, 1990.

[PATT94] Patterson, D. A. and Hennessy, J. L. *Computer Organization & Design: the hardware/software interface.* San Francisco: Morgan Kaufmann, 1994.

[QADA88] 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.

[RICH87] Richardson, J. P., et al. "Design and Evaluation of Parallel Pipelined Join Algorithms." *ACM SIGMOD*, Vol. 16, No. 3, Dec. 1987: 399-409.

[RITC74] Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *CACM*, Vol. 17, No. 7, Jul. 1974: 365-75.

[SCHN89] 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.

[SHAP86] 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.

[SHIN91] Shin, D. K. *A Comparative Study of Hash Functions for a New Hash-Based Relational Join Algorithm.* Pub. #91-23423, Ann Arbor: UMI Dissertation Information Service, 1991.

[SHIN94] Shin, D. K. and Meltzer, A. C. "A New Join Algorithm." *ACM SIGMOD RECORD*, Vol. 23, No. 4, Dec. 1994: 13-8.

[SHUL84] 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.

[SMIT79] Smith, D. C. and Smith, J. M. "Relational Database Machines." *IEEE Computer*, Vol. 12, No. 3, Mar. 1979: 18-38.

[STON76] Stonebraker, M. R., et al. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems*, Vol. 1, No. 3, Sep. 1976: 189-222.

[STON81] Stonebraker, M. R. "Operating System Support for Database Management." *CACM*, Vol. 24, No.7, Jul. 1981: 412-18.

[SU88] Su, S. Y. W. *Database Computers Principles, Architectures, and Techniques.* New York: McGraw-Hill, 1988.

[ULLM82] Ullman, J. D. *Principles of Database Systems.* Rockville: Computer Science Press, 1982.

[VALD82] Valduriez, P. "Semi-Join Algorithms for Multiprocessor Systems." *Proceedings of ACM's SIGMOD 1982 International Conference on Management of Data*, Jul. 1982: 225-33.

[VALD84] 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.

Relation S

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

Relation T

D	E	F
a	d	c
d	g	a
.	.	.
.	.	.
.	.	.

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

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

Relation R

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

Figure 1. An Example of the Join Relational Operation

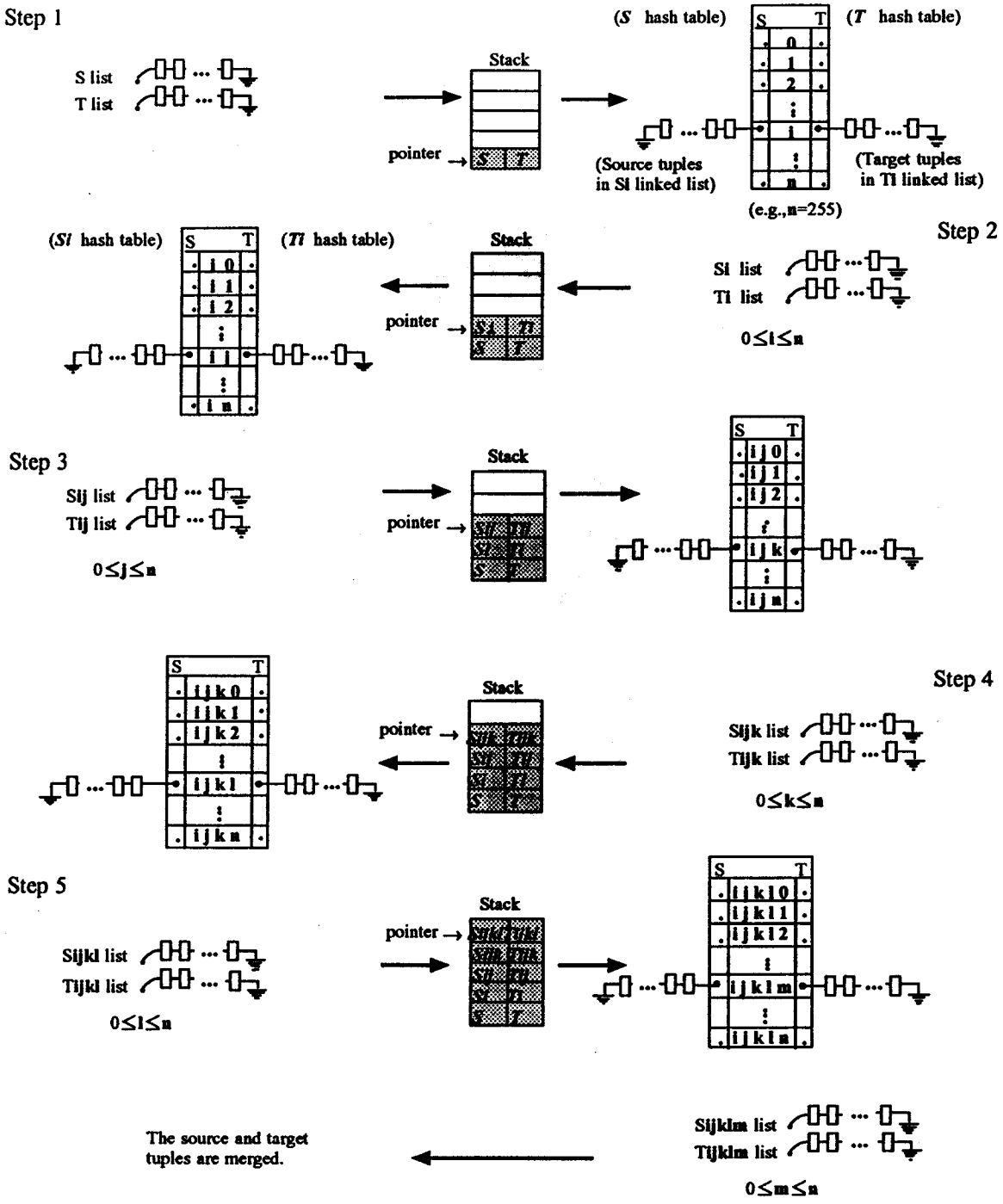

Figure 2. The SOFT and the Shin's Join Algorithm

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

Figure 3. The Shin's Join Algorithm

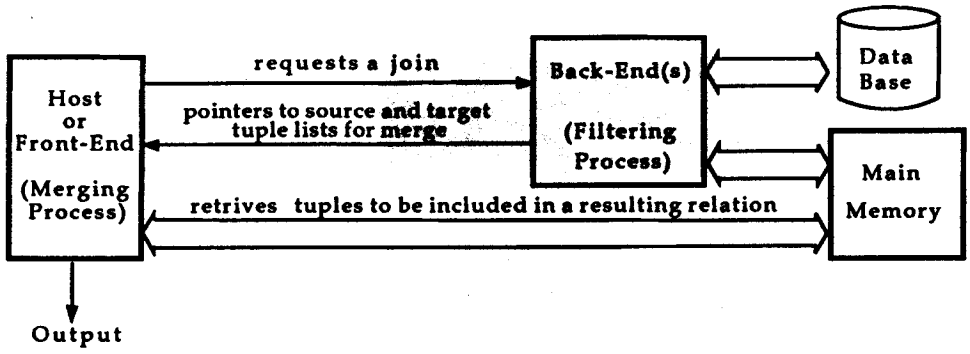

Figure 4. Shin's Join Method in HIMOD

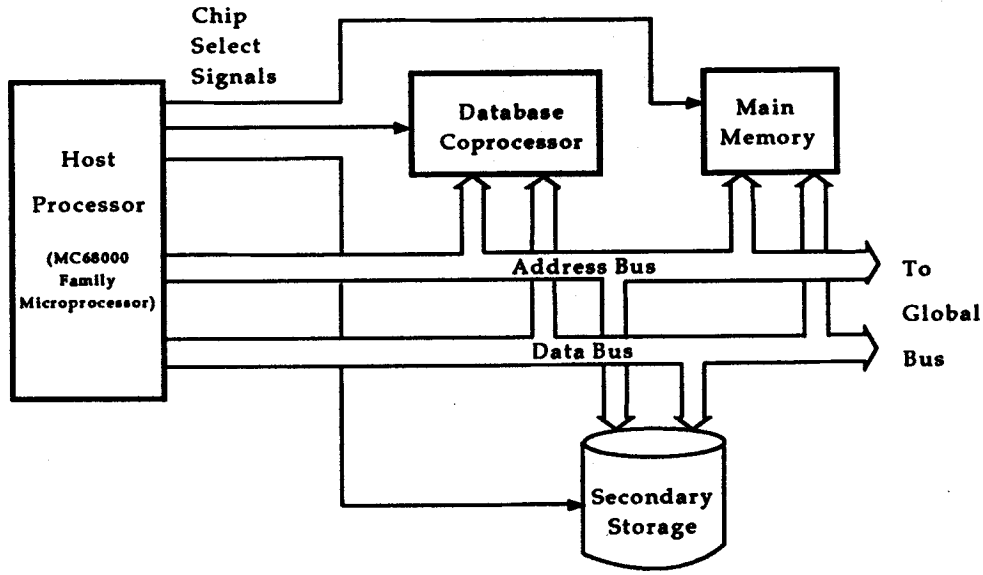

Figure 5. Coprocessor Configuration

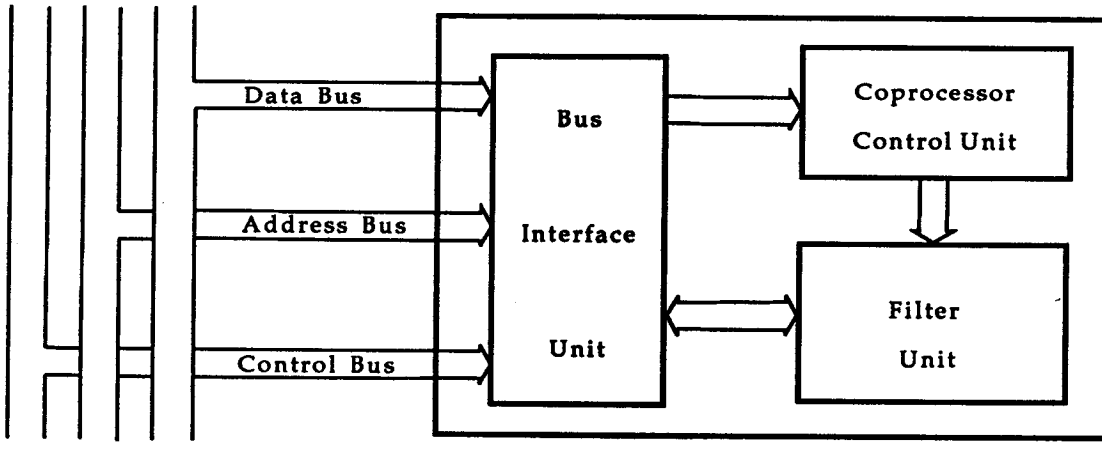

Figure 6. DBCP Simplified Block Diagram

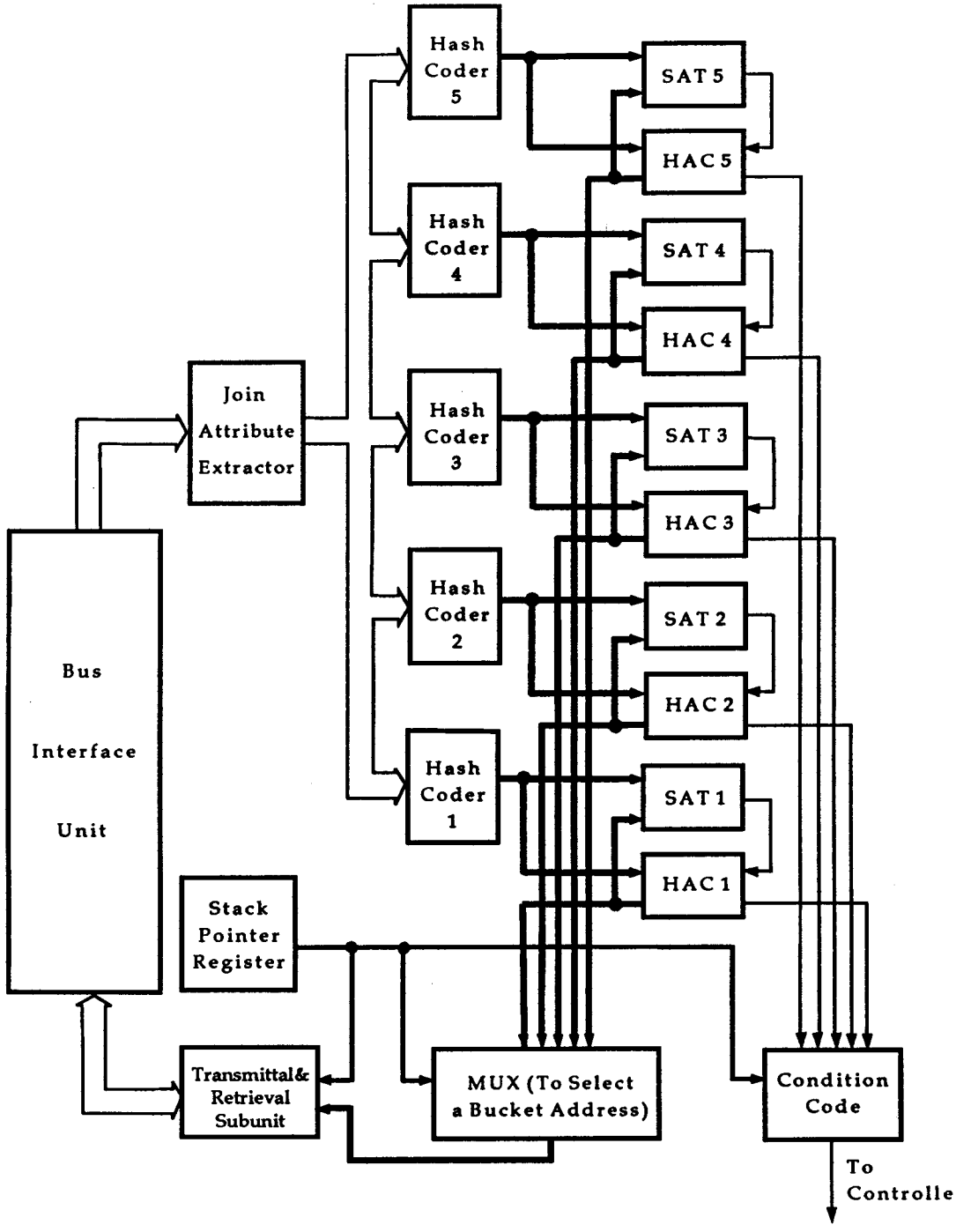

Figure 7. The DBCP Architecture (Filter Unit)

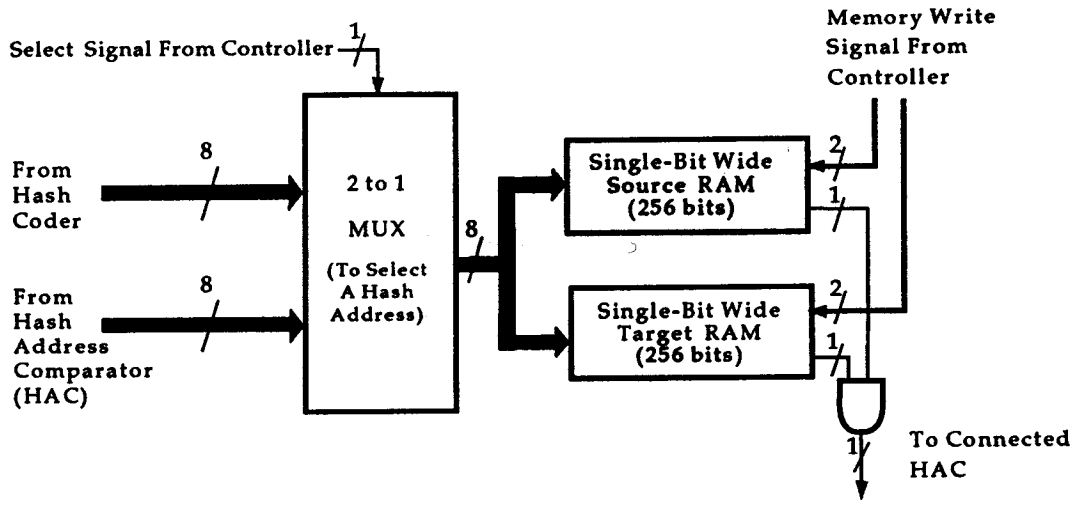

Figure 8. SAT (Source and Target Single-Bit Wide Memories)

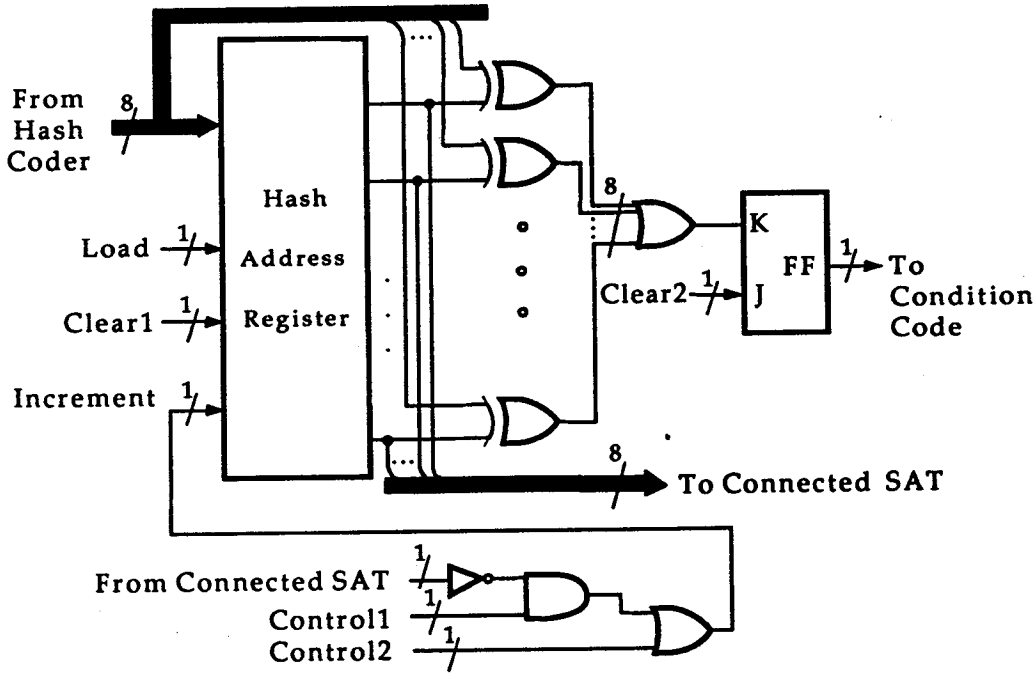

Figure 9. HAC (Hash Address Comparator)

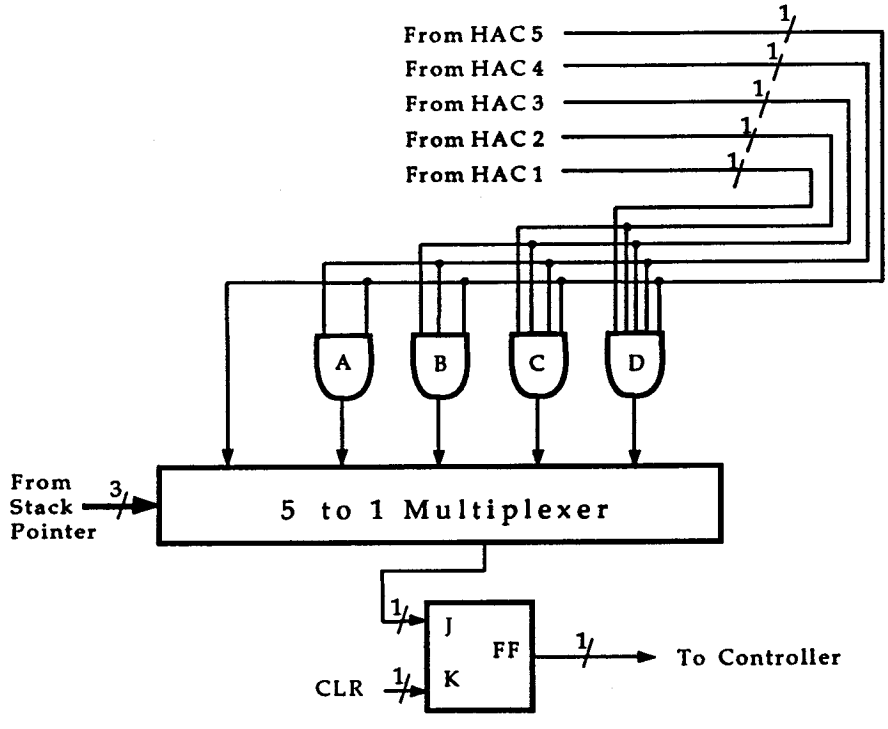

Figure 10. Condition Code for Checking if Only One Kind of Hash Address is Produced