New Generation Computing, 1 (1983) 63-74
OHMSHA, LTD. and Springer-Verlag

NEW GENERATION COMPUTING

© OHMSHA, LTD. 1983

Application of Hash to Data Base Machine and Its Architecture

Masaru KITSUREGAWA
Department of Electrical Engineering and Electronics, Institute of Industrial Science, The University of Tokyo, 22-1 Roppongi 7-chome, Minato-ku, Tokyo 106
Hidehiko TANAKA, and Tohru MOTO-OKA
Department of Electrical Engineering, Faculty of Engineering, The University of Tokyo, 3-1 Hongo 7-chome, Bunkyo-ku, Tokyo 113

Abstract In this paper we discuss the application of the dynamic clustering feature of hash to a relational data base machine. By partitioning the relation using hash, large load reductions in join and set operations are realized. Several machine architectures based on hash are presented. We propose a data base machine **GRACE** which adopts a novel relational algebraic processing algorithm based on hash and sort. Whereas conventional logic-per-track machines perform poorly in a join dominant environment, **GRACE** can execute join efficiently in $O\left(\frac{N+M}{K}\right)$ time, where N and M are the cardinalities of two relations and K the number of memory banks.

§ 1 Introduction

Many of the machines proposed so far, such as RAP[1] and SURE[2] have adopted as a base the 'Logic per Track' concept by Slotnick,[3] with some modification. This filter processing approach can reduce the amount of data that is transferred between the secondary storage device and main memory of a host computer. In supporting a relational data base, it outperforms conventional systems by several orders of magnitude in the execution of relatively light load operations such as selection and update. For heavy load operations, however, such as join and projection including duplicate elimination, we can only expect a slight improvement in performance.[4] There are other type data base machines,

such as DIRECT,[5] DBC join processor,[6] and RELACS,[7] but they adopt exhaustive matching algorithms to execute join and thus may not be sufficiently efficient in join, projection, and set operations for large relations.

In this paper, we discuss the application of hash to data base machines. It is shown that join and projection can be performed more efficiently through the utilization of the dynamic clustering feature of hash. Several machine architectures enhanced by the hash mechanism are presented. We have developed the data base machine **GRACE**, which adopts a novel relational algebraic processing strategy based on hash and sort.[8] Conventional cellular logic type data base machines composed of K cells execute join in $O\left(\frac{N \times M}{K}\right)$ time. **GRACE**, however, takes $O\left(\frac{N+M}{K}\right)$ time, where the relations with the cardinalities N and M are stored in K memory banks and $2 \times K$ processors are activated. Its architecture and bucket processing scheme are described. **GRACE** is expected to exhibit much higher performance in the join dominant environment than any data base machine proposed so far.

§2 Application of Hash to Data Base Machines

The first data base machine to utilize hash was CAFS.[9] Hash was used here as a joinability filter, where many tuples that cannot be joined are sieved out using a hashed bit array. This reduces the cardinalities of both relations and results in a large load reduction. While this method is very powerful for preprocessing, remaining tasks, such as the elimination of spurious tuples and tuple concatenation in explicit join, must be done on the host machine.

Besides the CAFS approach, the dynamic clustering feature of hash is also useful for heavy load operations such as join, projection, and set operations. In the clustering approach we adopted, it is not the number of tuples in two relations but the load of join itself which can be reduced. The ordinary join algorithm takes time proportional to the product of two relations' cardinalities. However, if two relations are clustered on the join attribute, that is, if the tuples are grouped into disjoint buckets based on the hashed value of the join attribute, there is no joining between tuples from buckets of different hashed value. Let

$$N=\sum_{i=1}^{s} n_i, \qquad M=\sum_{i=1}^{s} m_i$$

where N and M are the cardinalities of two relations, n_i and m_i are the sizes of the i-th bucket in each clustered relation, and s is the number of buckets. The total processing time T can be expressed as follows:

$$T \propto \sum_{i=1}^{s} n_i \times m_i$$

This load reduction effect is depicted in Fig. 1. The two axes denote two relations divided into s intervals, and the cross section reveals the join load of $n_i \times m_i$.

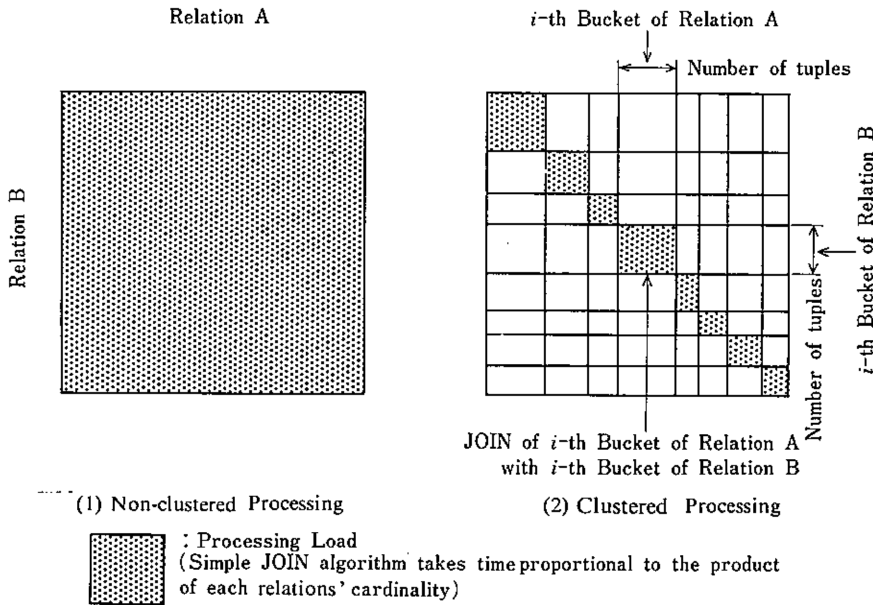


Fig. 1 Load reduction effects by clustering feature of hash in join operation

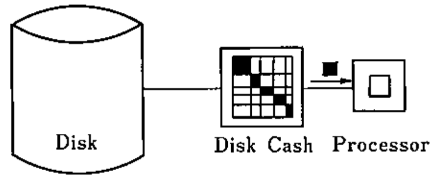
The shaded areas correspond to the processing load. Accordingly, this clustered approach can dramatically diminish the load in comparison with the nonclustered ordinary approach. The above approach can be applied to projection (duplicate elimination) as well as join. A data base machine utilizing this clustering approach would attain a very rapid relational algebraic execution, which we discuss in the following section.

§ 3 Classes of Machine Architecture Enhanced by Hash Mechanism

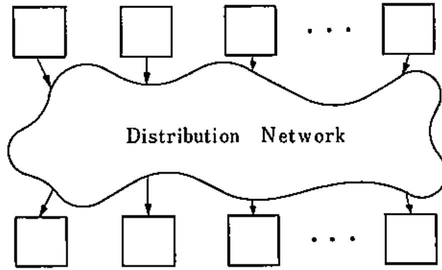
Several types of machine architecture can be configured to implement the hash-based relational algebraic processing scheme suggested in the previous section. Here we identify several classes.

3.1 Uniprocessor Architecture

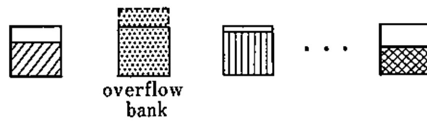
The clustering feature of hash can be fully utilized even in a conventional single processor environment. After generating a hashed relation in a disk cash as shown in Fig. 2 (a), a processor processes one bucket after another. Since the size of each bucket is rather small, a processor with an adequate local memory capacity can execute join of large relations efficiently. As described previously, the load of join and projection is reduced from $N \times M$ to $\Sigma\, n_i \times m_i$.



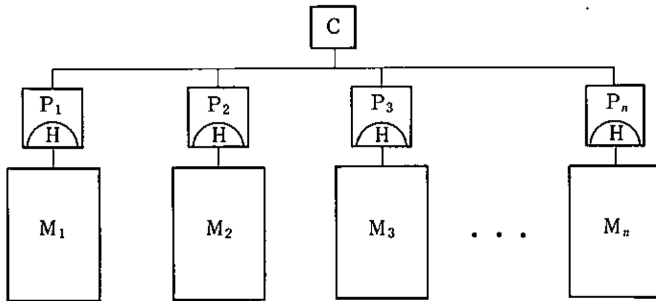
(a) Uniprocessor architecture



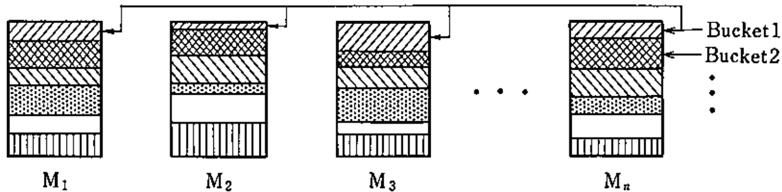
(b) Bucket converging architecture



(c) Abstract view of hashed relation on memory banks in bucket converging architecture



(d) Intra-bucket parallel processing architecture (Cellular logic type data base machine enhanced by hash mechanism)



(e) Abstract view of hashed relation spread over memory banks (Same pattern denotes tuples of the same bucket)

Fig. 2 Classes of machine architecture enhanced by hash mechanism

3.2 Multiprocessor Architecture

We can attain a higher performance for large data bases by storing them over multiple memory banks and processing buckets with many processors. From the mapping scheme of a logical view of hashed relations in multiple memory banks, we can see two approaches: a bucket converging architecture where all the tuples of a bucket are stored in only one memory bank, and a bucket spreading architecture where tuples of a certain bucket are stored over many memory banks.

[1] Bucket converging architecture

Before the execution of join, a relation stored over the source memory banks is hashed and distributed to destination banks so that all the tuples having the same hashed value converge into one bank (Fig. 2 (b)). Each destination bank has the logical view shown in Fig. 1 after the distribution. Since the tuples of one bank cannot be joined with those of another, processors dedicated to each bank can proceed independently in the same way as in uniprocessor architecture. In this architecture, however, the nonuniformity of hash causes memory bank overflow, where tuples gather in one bank beyond its capacity (Fig. 2 (c)). This phenomenon is the same as the bucket overflow found in the direct access method. It causes difficulties in memory management, such as determining the load factor, which is crucial in this architecture.

[2] Bucket spreading architecture

In bucket spreading architecture, there is no memory bank overflow. Instead it is necessary to provide some mechanism to process a bucket whose tuples are spread over many memory banks. Here we consider the following two approaches.

(i) Intra-bucket parallel processing architecture Many of the cellular logic type data base machines proposed so far can be much enhanced by introducing a hash mechanism (Fig. 2 (d)). Each processor hashes the tuples of its own memory bank on the join attribute. It adds a tag of hashed value to each tuple and restores it. A bucket is spread over some number of banks; the number of tuples of a certain bucket varies bank by bank. The logical view of a hashed relation is depicted in Fig. 2 (e), where the same pattern denotes the tuples of the same bucket. In this architecture, join can be executed in the same way as RAP.[1] Namely, tuples of a selected source bank are broadcast to all target banks, where each processor joins the distributed tuple with its own tuples. A processor need not compare a received tuple with all those in its own bank, but rather only with those with the same hashed value. Therefore, fair performance improvement can be expected if the machine employs a storage medium where the retrieval time of a bucket's tuples is almost proportional to the volume of the bucket. One track of a disk is not adequate because it always takes one revolution to retrieve any amount of records. For example, machines employing major/minor magnetic bubble memory instead of disk such as EDC[10]

would show higher performance. This is because the bubble memory is a pseudo-random access device and can skip unnecessary records rapidly. We discuss the enhanced bubble chip organization in 4.1 [2]. When a source bank finishes broadcasting its own tuples, the next bank starts in the same way. After all the tuples of the bucket over all source banks are sent to the destination bank, the next bucket processing begins. Thus cellular logic type data base machines can execute join much more efficiently by introducing a hash mechanism than the conventional cellular logic type architecture. Each bucket is processed in parallel by many processors, and buckets are handled serially. This architecture is characterized by intra-bucket parallel processing and inter-bucket serial processing. Machines of this type also have some problems. Since the subbucket sizes are different in different banks, some banks might take a long time to process their subbuckets, while some are idle. Here we call the portion of the bucket contained in each bank a subbucket. All subbuckets spread over some number of banks amount to one bucket. The bank of the largest subbucket might decrease the performance of bucket processing.

(ii) Pipelined bucket processing architecture In contrast with the previous architecture, a bucket here is processed by only one processor. Several buckets are processed simultaneously by activated multiple processors. In this sense it has the same features as bucket converging architecture, but there is no bank overflow, and the processing scheme is quite different.

A processor gathers the tuples of the allocated bucket from the memory banks. This gathering process proceeds in pipeline fashion. That is, a processor visits bank one after another and inputs the necessary tuples, while memory banks output the tuples of the bucket allocated to that processor. A processor runs through a pipe composed of memory banks. After it inputs all the relevant tuples, it begins relational algebraic processing. Multiple processors flow in a pipe, corresponding to a data stream in an ordinary pipeline system. Since a bucket is allocated only to one processor, there is no inter-processor communication. Each can proceed independently. When a processor finishes one bucket, it enters the pipe again. This architecture is characterized by inter-bucket parallel processing and intra-bucket serial processing. Of course, buckets beyond the available processors are processed serially. We discuss this architecture in more detail in the next section.

§4 Architecture of GRACE

In the previous section, several machine architectures utilizing hash were introduced. We designed a relational algebra machine **GRACE** belonging to the last category, pipelined bucket processing architecture.

4.1 Bucket Processing in GRACE

In **GRACE** the buckets generated through hash are processed by dynamically allocated processors in parallel. First we clarify the bucket processing

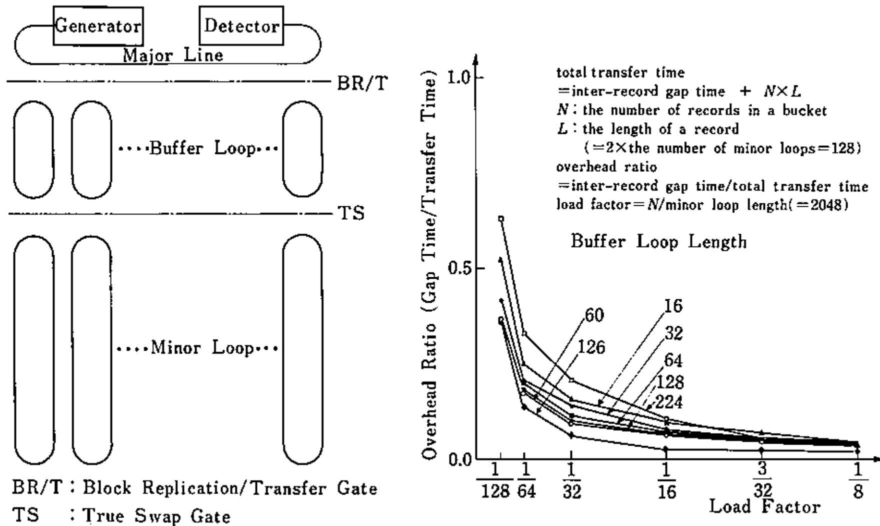
mechanism of **GRACE** : how to process individual buckets in a processor, how to allocate buckets to processors, how to control the bucket stream without disturbing the pipeline, and how to manage the buckets in the memory banks. Thereafter, the actual architecture is presented.

〔1〕 Bucket processing based on sorting

We have not yet discussed how to process an individual bucket. There are a number of methods. Since the size of one bucket is relatively small, ordinary exhaustive matching is possible. Associative array or parallel comparison logic which many of the proposed machines adopted is another candidate. In order not to disturb the pipeline, it is necessary to process a bucket in $O(n)$ time, where n is the size of a bucket. The first method takes time proportional to the product of the cardinalities of two relations in join. The second makes on-the-fly processing possible, but the array capacity is limited in current technology. We adopted a hardware sorter, based on the pipeline marge sort,[11] which completes sorting in $O(n)$ time and has less of a capacity problem. It can output a sorted data stream with a very small delay after inputting all the records. It is obvious that a relational algebraic operation can be executed in $O(n)$ time if the relation is sorted, but the sorter cannot complete sorting until the last data item arrives. Therefore, it takes longer to process a bucket than on-the-fly processing. But this hardly affects performance because buckets are processed in pipeline fashion, as discussed in the following paragraphs.

〔2〕 Bucket output mechanism

Memory banks are required to generate an efficient bucket serial data stream to processors. We find semiconductor RAM and magnetic bubble memory in the memory hierarchy as the storage medium of the bank. Here we consider bubble based organization. The pseudo-random accessibility of major/minor bubble memory is the key feature in efficient bucket serial data stream generation. The unnecessary records stored over minor loops can be skipped in a period of 1 bit field rotation time. This quick skip mechanism improves the access time and effective transfer rate of bubble memory. In pipelined bucket processing architecture, memory banks need to output the tuples of a given bucket contiguously. Rather than an individual record access, a set oriented access should be supported efficiently, sequence of the records doesn't matter. Therefore, the effective transfer rate is more influential than the access time. We modified a conventional major/minor bubble chip to get a higher transfer rate. As is shown in Fig. 3 (a), short buffer loops are inserted between major lines and minor loops. Necessary records can be buffered in the added loops while a record is being output bit-serially. Block replication/transfer gates are used between major lines and buffer loops, and swap gates between buffer loops and minor loops. Swap gates permit look-ahead buffering. The gates are controlled by a bubble control unit using a mark-bit RAM which synchronizes the magnetic field rotation. The hashed value of a tuple is stored in this RAM. The gates are controlled so that tuples with the same hashed value are output



(a) Chip organization (b) Overhead ratio of the modified bubble chip

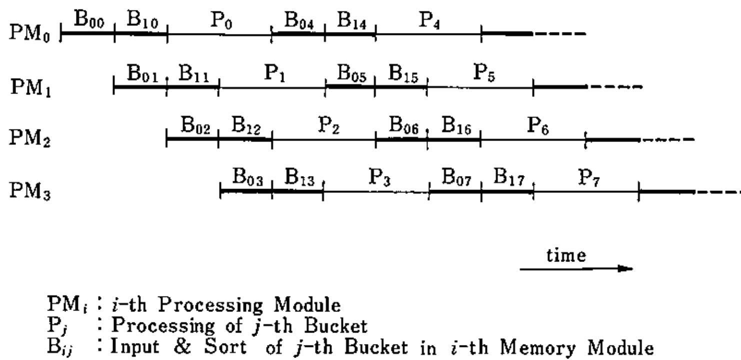
Fig. 3 Modified major/ minor bubble chip with buffer loops

continuously. Figure 3 (b) indicates the simulation results of the performance of this chip. Buffer loops make the inter-record gap time almost negligible ; that is, efficient bucket serial data stream generation is realized.

〔3〕 Bucket processing pipeline

Here we discuss the pipelined processing of the buckets in more detail. With the adoption of a hardware sorter, it takes about $2 \times n$ time for a processor to handle a bucket of n tuples. That is, it takes n time to gather the tuples from the memory banks and another n time for relational algebraic operation on the sorted tuples. It should be noticed that tuple gathering and sorting are overlapped, and the sorter can begin to output the sorted data stream when it inputs the last tuple. Memory banks continuously output the tuples of the bucket considered. Tuple gathering is done by pipelined processing, as mentioned in 3.2〔2〕(ii). D_{ij} denotes the tuples of j-th bucket stored in the i-th memory bank. Figure 4 shows the pipeline processing overview, where the number of memory banks and processors are two and four, respectively. Thus $2 \times K$ processors are activated to process the data streams generated by K memory banks. Here we assume that D_{ij} is the same size for all i and j.

Pipeline processing works well when each segment time is equal. If a segment takes a long time, it becomes a bottleneck. In actual processing we cannot expect an ideal case like Fig. 4. Both the volume of buckets and number of tuples belonging to a certain bucket differ greatly among banks, meaning that the segment time of the pipeline varies dynamically. This causes pipeline disturbance and results in performance degradation. To resolve this segment



PM_i : i-th Processing Module
P_j : Processing of j-th Bucket
B_{ij} : Input & Sort of j-th Bucket in i-th Memory Module

Fig. 4 Overview of pipelined bucket processing

time fluctuation, we have to spread the tuples of a bucket almost evenly over memory banks. Accordingly, we perform 'bucket flat distribution', through which a tuple of the j-th bucket emitted by a source bank is controlled to fall into the i-th destination bank, where the size of D_{ij} is minimum among the banks at that time.

Nonuniformity of the hash function is inevitable. Therefore, it is difficult to make the sizes of buckets equal. From the point of processor utilization, it is desirable that the size of a bucket be close to the processor's capacity. Thus, we do 'bucket size tuning' : First we partition the relation into smaller buckets, then integrate some of these into a larger bucket whose volume is less than the capacity of a processor. Through this tuning, buckets with the size of near uniformity can be produced.

4.2 Hardware Architecture of GRACE

The global architecture of **GRACE** is shown in Fig. 5. **GRACE** consists of four kinds of fundamental modules : processing, memory, disk, and control modules. These are connected through a processing ring and staging ring. The relations stored in disk modules are staged into memory modules and processed by processing modules through the processing ring. The time divison multi-channel ring buses make many modules run simultaneously. We shall describe each component briefly.

A processing module handles an allocated bucket with a hardware sorter, as was discussed in 4.1[1]. The $O(n)$ sorter can sort the tuple stream generated by memory modules keeping up with the stream. We have implemented a sorter with microprogrammed control and attaind a processing rate of 3 Mbyte/sec,[12] comparable to the highest transfer rate of the current disks. A tuple manipulation unit, another important component, performs relational algebraic operations on the sorted tuple stream. The third component in a processing module is a hashing unit, which hashes the tuple over the attributes

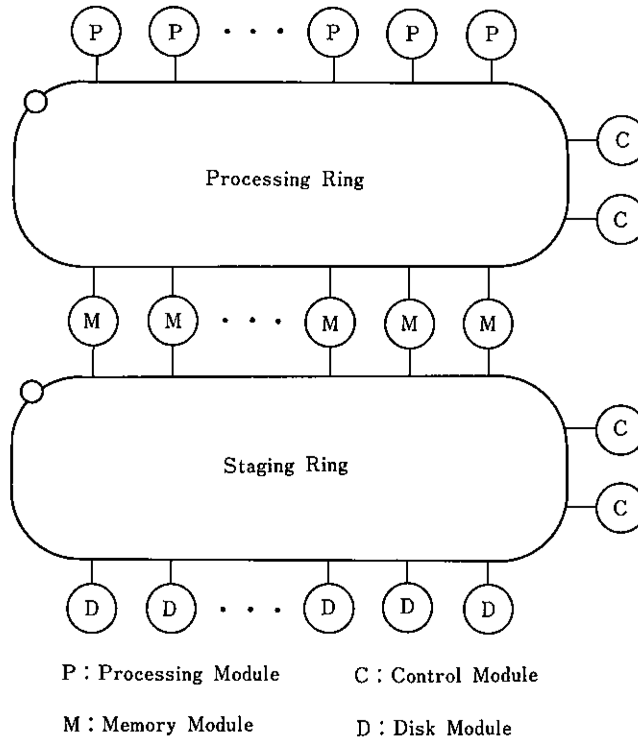


Fig.5 Global architecture of **GRACE**

for the next operation.

A memory module provides a large working space for intermediate and final relations and generates an efficient bucket serial data stream to processing modules. We now use bubble memory to construct this, as discussed in 4.1 [2]. A modified bubble chip with buffer loops is available at present.[13] The major drawback in the current magnetic bubble memory is its low transfer rate. This problem can be resolved by activating multiple chips in parallel. We have designed a pilot module incorporating 16 chips.

A disk module literally adopts a disk as the secondary storage to support large data bases. In staging relations to memory modules, filter processors in disk modules evaluate the simple predicates on the fly and eliminate unnecessary records. The irrelevant attributes are also removed. This mechanism is found in many data base machines such as RAP and TIP in DBC, and reduces the amount of data to be transferred between modules. There is also a hashing unit, which adds the hashed value of join or projection attributes to the tuple.

§5 Query Execution on GRACE

Here we describe query execution on **GRACE** architecture. First the necessary relations are staged from disk modules to memory modules through a staging ring. The data streams from the disks are filtered out and hashed values are added. Only necessary tuples are transferred to memory modules, where they are stored in bubble memory and tags of hashed value in a corresponding mark-bit RAM. When the data transfer is completed, a hashed relation is generated conceptually on memory modules. Then an actual relational algebraic processing is performed through a processing ring. Memory modules generate bucket serial data streams, and processing modules gather the tuples of the allocated bucket. This proceeds in pipeline fashion as shown in Fig. 5. Once a processor is finished with one bucket, it begins with another. The relational algebraic operation ends when all the buckets are consumed. The data streams generated in memory modules are hardly disturbed. Thus, join is executed in $O\left(\frac{N+M}{K}\right)$ time, where N and M are the cardinalities of two relations and K the number of activated memory modules.

For a complex query comprising many joins and/or set operations, the result relation produced in memory modules is joined with another relation. Recall our processing strategy for join. First the relation is hashed into small buckets, and thereafter the generated buckets are processed. Therefore, it seems that the hashing operation must be interleaved for each intermediate relation. But there is no need for such an extra processing phase. A processing module hashes result tuples while processing a bucket. That is, hash processing of a result relation for the next operation can be overlapped with the actual processing of the current operation. We call this 'operator level pipeline'. This eliminates the hashing overhead time. Thus, **GRACE** can execute a complex query very efficiently.

§6 Conclusion

We have examined various approaches to utilizing the clustering feature of hash in data base machines and introduced possible architectures. Through partitioning the relation using hash, dramatical performance improvement in join and set operations can be attained. **GRACE** adopts a novel relational algebraic processing strategy based on hash and sort, where hashing overhead is canceled by the operator level pipeline effect. **GRACE** is expected to execute a complex query comprising many joins and/or projections much faster than any data base machine proposed so far. The details of **GRACE** and the performance evaluation will be reported in a future paper.

References

1) Ozkarahan, E. A., Shuster, S. A. and Smith, K. C.: RAP-An Associative Processor for Data Base Management, Proc. AFIPS NCC, *45* (1975) 379.
2) Leilich, H. O., Steige, G. and Zeidler, H. C. H.: A Search Processor for Data Base Management Systems, Proc. Int. Conf. on VLDB (1978) 280.
3) Slotnick, D. L.: Logic per Track Devices, Advances in Computers, *10* (Academic Press, New York, 1970) 291.
4) Ozkarahan, E. A., Shuster, S. A. and Smith, K. C.: Performance Evaluation of a Relational Associative Processor, ACM Trans. Database Syst., *2* (1977) 175.
5) DeWitt, D. J.: DIRECT-A Multiprocessor Organization for Supporting Relational Database Management Systems, IEEE Trans. Comput., *C-28* (1979) 395.
6) Menon, M. J. and Hsiao, D. K.: Design and Analysis of a Relational Join Operation for VLSI, Proc. Int. Conf. on VLDB (1981) 44.
7) Oliver, E. J. and Berra, P. B.: RELACS-A Relational Associative Computer System, Proc. of the Fifth Workshop on Computer Architecture for Non-Numeric Processing (1980) 106.
8) Kitsuregawa, M., Suzuki, S., Tanaka, H. and Moto-oka, T.: Relational Algebra Machine based on Hash and Sort, *TGEC81-35* (IECE Japan, 1981).
9) Babb, E.: Implementing a Relational Database by Means of Specialized Hardware, ACM Trans. Database Syst., *4* (1979) 1.
10) Uemura, S., Yuba, T., Kokubu, A., Ooomote, R. and Sugawara, Y.: The Design and Implementation of a Magnetic Bubble Database Machine, Proc. of IFIP Congress (1980) 433.
11) Todd, S.: Algorithm and Hardware for a Merge Sort Using Multiple Processors, IBM J. Res. & Dev., *22* (1978) 509.
12) Kitsuregawa, M., Fushimi, S., Kuwabara, K., Tanaka, H. and Moto-oka, T.: An Organization of Pipeline Merge Sorter, Trans. IECE Japan, *J66-D* (1983) 332 [in Japanese].
13) Kohara, H., Takahashi, K., Suga, S. and Fujiwara, S.: Novel megabit bubble memory chip organization and its characteristics, Proc. of ICMB-4, *D-3* (1980).