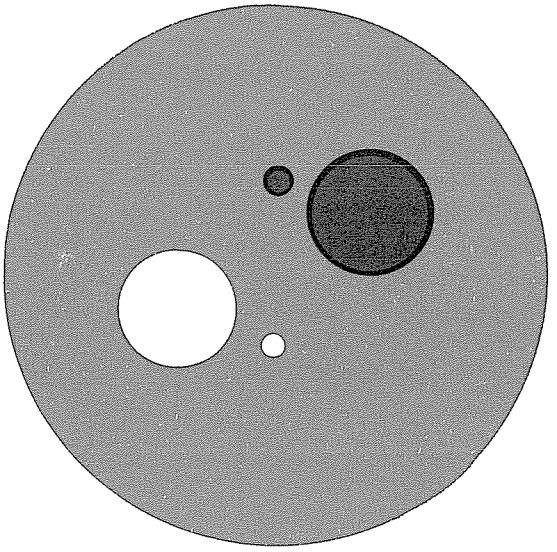
COMPUTER SCIENCES
DEPARTMENT

University of Wisconsin-
Madison



DESIGN CONSIDERATIONS FOR DATA-FLOW DATABASE MACHINES

by

Haran Boral
David J. DeWitt

Computer Sciences Technical Report #369

October 1979

Design Considerations for Data-flow Database Machines

Haran Boral
David J. DeWitt
Computer Sciences Department
University of Wisconsin
Madison, Wisconsin
USA

This research was partially supported by the National Science Foundation under grant MCS78-01721 and the United States Army under contracts #DAAG29-79-C-0165 and #DAAG29-75-C-0024.

ABSTRACT

This paper presents a discussion of the application of data-flow machine concepts to the design and implementation of database machines which execute relational algebra queries. Three levels of operand granularity for data-flow database machines are introduced and compared. We demonstrate, that relation-level granularity is too coarse and that tuple-level granularity is too fine. The third level of granularity, a page of a relation, is shown to be the best choice from both hardware and software viewpoints. Finally a preliminary design for a data-flow database machine which utilizes page-level granularity and supports distributed control of instruction execution is presented.

1.Ø Introduction

During the past several years we have been investigating the design and implementation of multiprocessor database machines for the execution of relational algebra queries. In [1,2] the architecture of DIRECT, a MIMD database machine, is described. The problem of relation fragmentation and its impact on query execution time is discussed in [3]. In [4], four processor assignment strategies for MIMD database machines are described and evaluated. One of the primary results presented in [4] is that the application of data-flow machine techniques to the processing of relational algebra queries significantly enhances system performance. However, it leaves open several questions about data-flow database machines which we intend to address in this and future papers.

In Section 2.Ø, we introduce the basic concepts of query processing and data-flow machines. In Section 3.Ø, three levels of operand granularity for data-flow database machines are introduced and compared. We will demonstrate, that relation-level granularity is too coarse and that tuple-level granularity is too fine. The third level of granularity, a page of a relation, is shown to be the best choice from both hardware and software viewpoints. Then, in Section 4.Ø, a preliminary design for a data-flow database machine which supports page-level granularity is presented.

The architecture of [1,2] is that of a data-flow machine

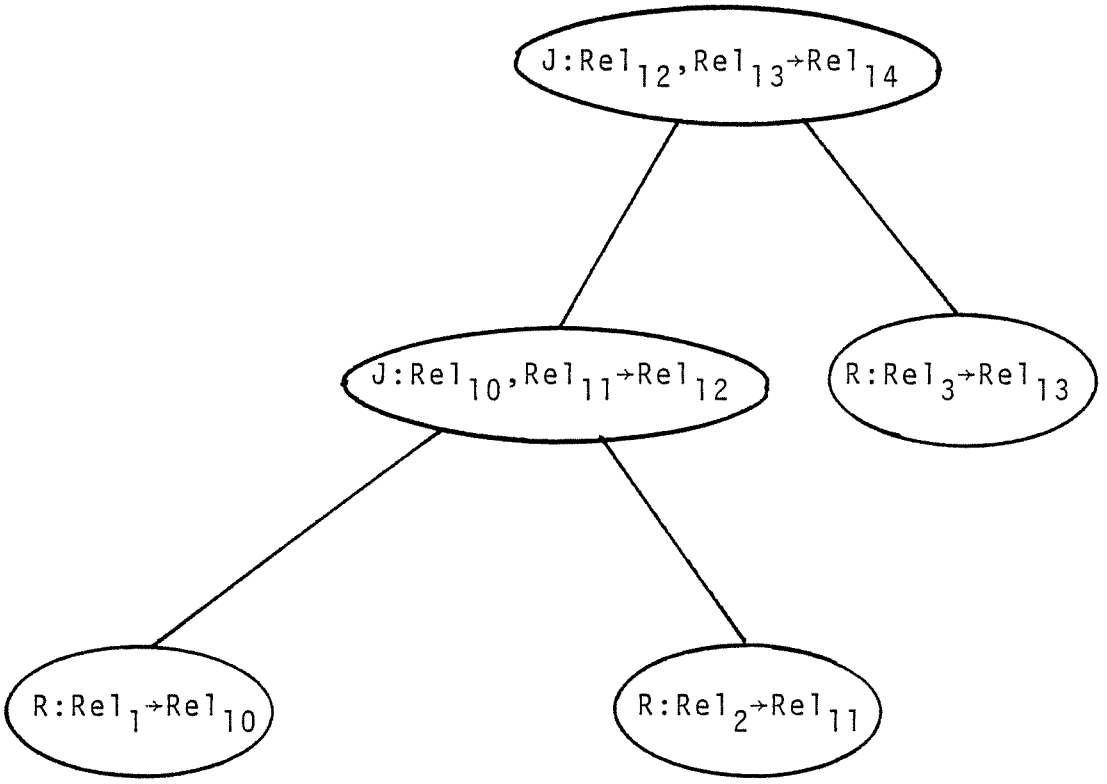
where all the control functions are centralized. We would like to design a machine with distributed control. Past work [3,4] has strongly suggested that there are a large number of hidden problems. Consequently we feel that one is well advised to move towards a design with distributed control in several steps. This paper is our first such step.

2.0 Background

2.1 Relational Query Processing

Each relational algebra query is generally comprised of one or more relational algebra operations (instructions) and is organized in the form of a tree. Each node, represents an operation to be performed on a number of relations. Some examples are restrict, join, append, and delete. Nodes higher up in the tree operate on relations computed by nodes below them. Figure 2.1 contains an example of a typical relational algebra query in the form of a query tree.

In a relational database system one of the most time consuming operations that must be performed is the join operator (a conditional cross product of two relations). In [5], several alternative join algorithms for uniprocessor systems are presented and analyzed. The computational complexity of each ranges from O(nlogn) for the "sorted-merge" algorithm to $O(n^2)$ for the "nested-loops" algorithm. While the nested-loops join algorithm is the slowest on a single processor, it appears to be the best



R:Restrict
J:Join

FIGURE 2.1

A SAMPLE QUERY TREE

algorithm for execution of the join operator on multiple processors. The algorithm works by joining each of n entities in one relation (the outer relation) with all of the entities in the other relation (the inner relation). If we consider an entity to be a tuple and there are n processors available, then each processor can join one tuple of the outer relation with the entire inner relation. Therefore, the execution time will be 1/n th the time required for a single processor to execute the join.

All the other algorithms presented in [5] involve performing some operation on the entire relation (sorting it or creating an index on some attribute). Although these algorithms can be performed on a multiple processor system, they are difficult to implement and at various points severely constrain the amount of parallelism that can be exploited.

2.2 Data-flow Machines

A data-flow machine is an architecture devoid of a program counter where instructions are enabled for execution as soon as their operands are present. Such a machine consists of a memory section, a processing section, and an interconnection device between the two sections. A memory cell contains an instruction and room for the operand data. As soon as all the required data is present, the contents of the cell are sent to some processor for execution. This frees the cell for the execution of the next instruction. Output from the processor is sent via the interconnection device to one or more memory cells, possibly enabling one

or more instruction(s) in the destination cell(s).

Various architectures for data-flow machines have been proposed [6-10]. These architectures differ from each other in many ways. One difference is the granularity of the operands and the types of operations that the processors execute. For example, Dennis [6] talks about assigning such instructions as add and multiply to the processors whereas Arvind [7] and Rumbaugh [9] assign entire procedures to processors.

For data-flow database machines there are also several alternative variable granularities for enabling relational algebra operators in the query tree. That is, the basic variable used for scheduling decisions can be a whole relation, a fragment of a relation, or a single tuple. In Section 3.0, we will describe and then contrast each of the these granularities.

In order to illustrate our ideas we chose to use the MIT machine [6] as a model since it is easy to understand and describe. Furthermore we feel that although the model differs significantly from others the basic results remain unchanged. The machine organization of Figure 2.2 depicts the model described in [6]. Although later, more sophisticated, variations have been described in the literature [10] we feel that they do not conceptually differ from the original.

In the machine of Figure 2.2 the interconnection mechanism is divided into two sections. The arbitration network provides a path from every memory cell to every processor. Enabled cells travel through it to processors for execution. Result packets

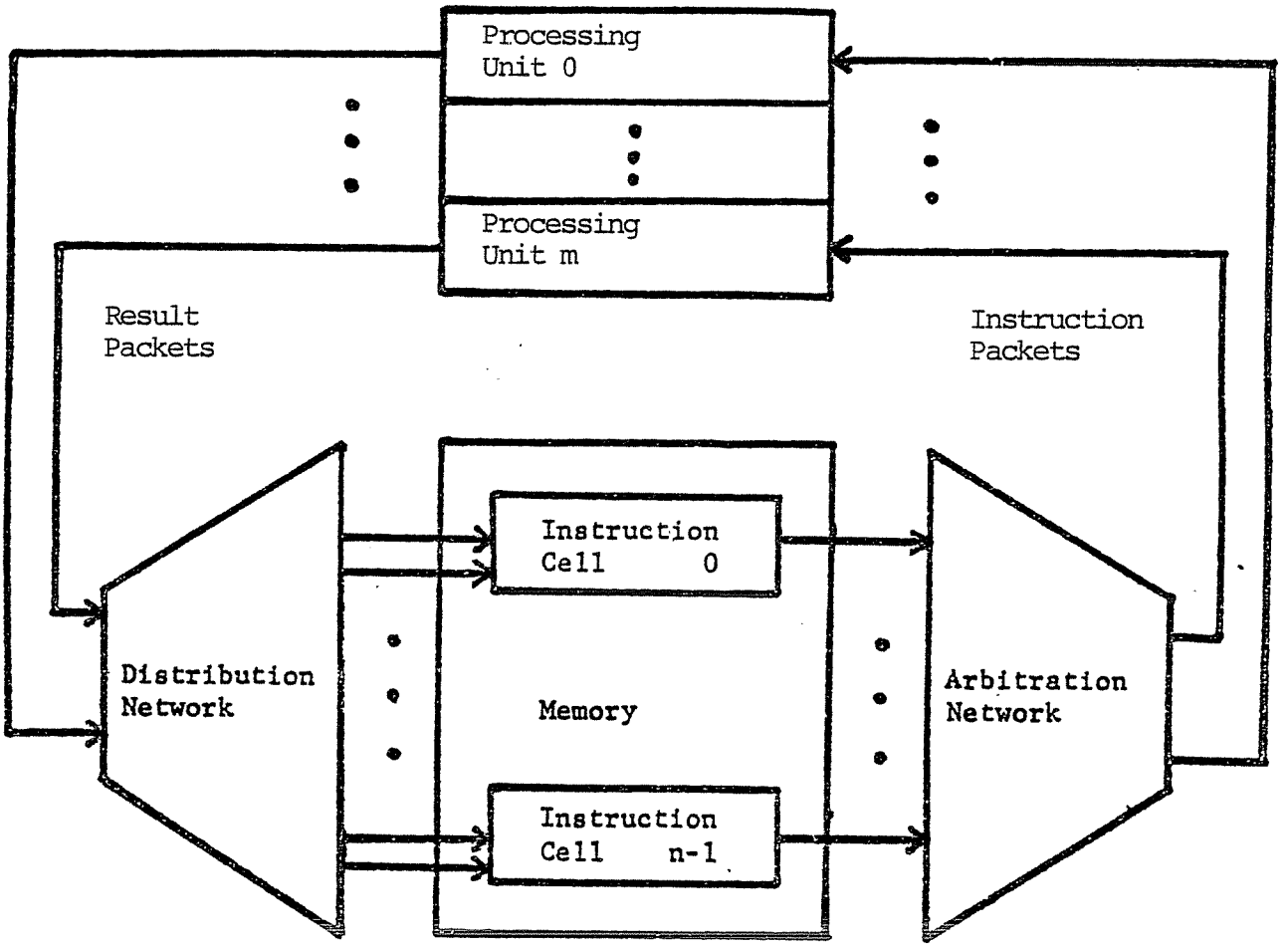


FIGURE 2.2

THE MIT DATA-FLOW MACHINE MODEL

are sent from the processors through the distribution network to the memory.

2.3 Relational Query Processing in Data-Flow Machines

We assume that the instruction in each memory cell corresponds to a node in the query tree and that the data is represented by page tables, pointing to pages either in a cache or on mass storage. Thus a relation can also be thought of as a stream [10] of pages (there is a close correspondence between the stream operators of [10] and the page operations of [1,2]). In order to simplify our discussion we assume that at the time that a memory cell fires, the associated data pages are retrieved from a cache and placed, together with the control information, on the arbitration network. Similarly, the distribution network places output pages in the cache and updates the page tables in the target cells.

The processing of queries in a data-flow fashion is related to the idea of processing relational queries in a pipelined fashion which has been previously suggested by Smith and Chang [11] and Yao [12]. There are, however, several important differences between the two approaches. The first deals with the number of processors which can be used to execute each node in the query tree. In the pipelined approach, there will be at most one processor executing each node in the tree and therefore the concurrency obtained will be limited by the number of nodes in the query tree. In the data-flow approach we can have any number

of processors executing each node and can dynamically adjust which processors are executing which nodes in the query tree in order to maximize performance. The other major difference is that in the data-flow approach we never need to wait for one node to completely finish before initiating the subsequent operator as has been suggested is necessary for pipelining [12].

3.0 Three Operand Granularities for Data-flow Query Processing

3.1 Relation-level Granularity

The coarsest possible granularity for enabling instructions is the relation. That is, a node in the query tree is enabled for execution only when its source operand(s) has (have) been completely computed. Clearly, if the query is in a tree format, all leaf nodes are immediately executable. A node higher up in the query tree is enabled whenever all of its descendants have finished executing.

3.2 Page-level Granularity

This approach is a variation of the relation-level granularity except for the criterion which is used to enable instructions in the query tree. In this approach a page of a relation (containing a set of tuples) is used for scheduling decisions. This means that an operator can be initiated as soon as at least one page of each participating relation(s) exists. Assigning processors to operate on pages rather than relations offers the possi-

bility of having a very flexible processor allocation strategy (that is, an instruction's "needs" are evaluated more often). Furthermore, it becomes possible to cut down on page traffic between the data-flow machine memory and the mass storage device(s) by distributing processors across all nodes of the tree and pipelining pages of intermediate relations between them.

In order to evaluate data-flow query processing which employs relation-level granularity with page-level granularity a simulation was implemented. While this simulation measures the performance of each data-flow strategy on a multiprocessor organization [1,2] which is not a true data-flow machine (i.e. it has centralized control), we feel that similar results would be obtained if the strategies were tested on a machine with more decentralized control organization.

Using a benchmark containing ten queries (2 queries with 1 restrict operator only, 3 queries with 1 join and 2 restricts each, 2 queries with 2 joins and 3 restricts each, 1 query with 3 joins and 4 restricts, 1 query with 4 joins and 4 restricts, and 1 query with 5 joins and 6 restricts), a relational database containing 15 relations with a combined size of 5.5 megabytes, and two memory cells for each processor, these two granularities were compared. The results are presented in Figure 3.1. As illustrated by this experiment (and for 5 other tests presented in [4]), the page-level granularity generally outperforms relational-level granularity by a factor of about two to one. These results seem to verify the benefits of pipelining pages of

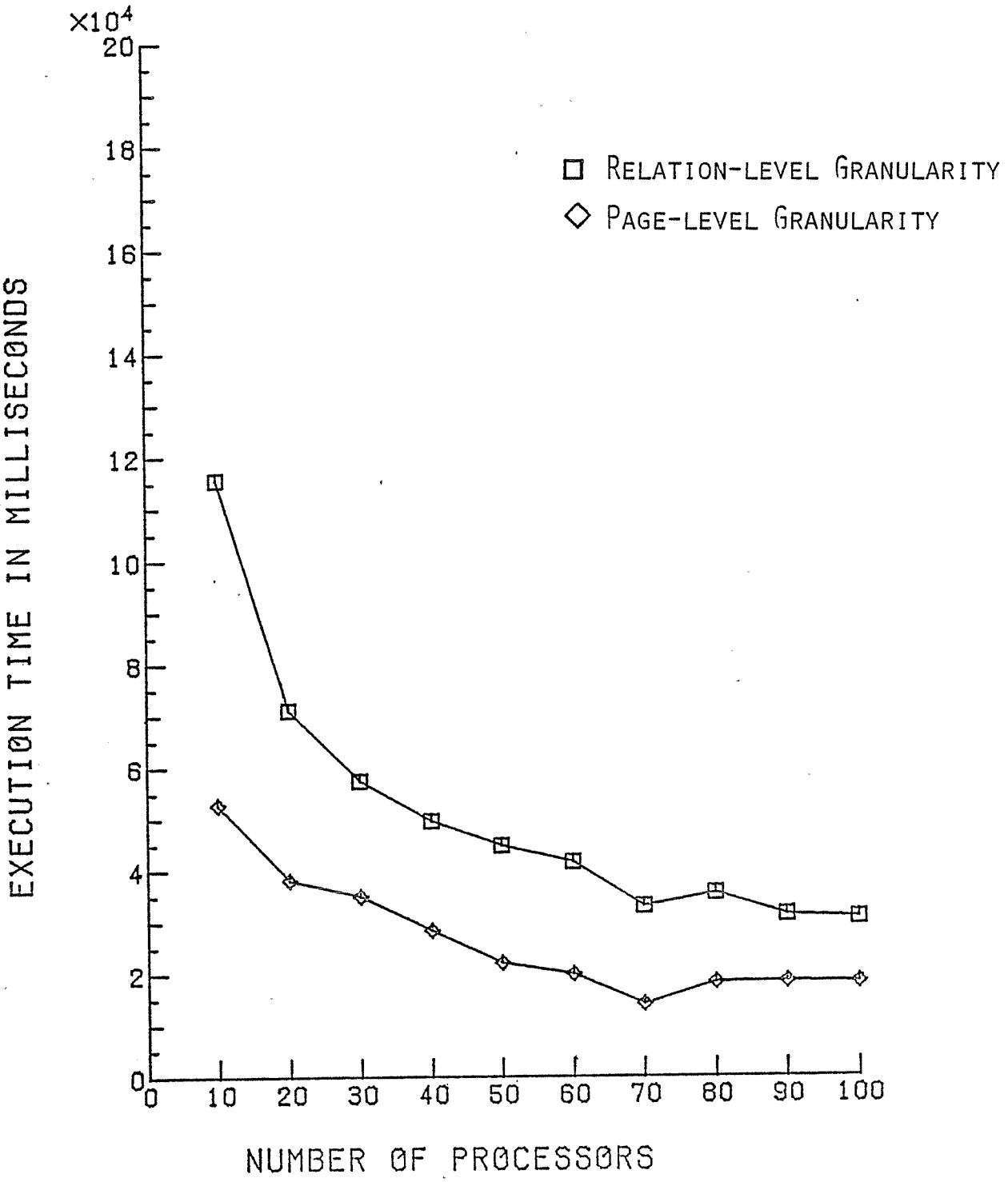


FIGURE 3.1

COMPARISON OF PAGE-LEVEL AND RELATION-LEVEL GRANULARITIES

relations up the query tree in order to minimize movement of data between a shared data cache and secondary memory and, therefore maximize system performance.

3.3 Tuple-level Granularity

In this approach a tuple of a relation is the basic unit which is used for scheduling decisions. This means that an operator can be initiated as soon as at least one tuple of each participating relation(s) exists. As with page-level granularity, this granularity also offers the possibility of having a flexible processor allocation strategy and pipelining tuples of intermediate relations between nodes in the query tree.

There appear to be two primary hardware limitations of the tuple-level granularity: the bandwidth requirements placed by such an approach on the arbitration network and the movement of data from the primary memory of the data-flow processor to mass storage.

When the nested-loops join algorithm is applied with tuple-level granularity, each tuple of the "outer" relation will be joined with every tuple of the "inner" relation. Let the outer relation be A and the inner relation be B. Assume that the number of tuples in A is n and the number of tuples in B is m. Furthermore, assume that each tuple in A and B is 100 bytes long and that c represents the number of overhead bytes associated with each instruction that passes through the arbitration network. Therefore to execute the join, n*m*(200+c) bytes will have

to pass from the memory through the arbitration unit to the processing section.

Next consider the bandwidth requirements if this same example is executed using page-level granularity. Assume that each page is 1000 bytes long. Therefore, relation A occupies n/10 pages and relation B occupies m/10 pages. The number of bytes that must pass through the arbitration unit is thus n/10 * m/10 * (2000 + c), which reduces to n*m*(20 + c/100) bytes. Even if one ignores the overhead of sending a packet (which is probably the same for both granularities), the bandwidth requirements of the page approach is 1/10 that of the tuple level approach.

While increasing the page size to 10,000 bytes will obviously decrease the arbitration network bandwidth requirements by another order of magnitude, such an increase may have an adverse effect on query execution time because it may reduce the maximum degree of concurrency which is possible. If, for example, the number of processors available for query execution is approximately equal to n * m, then tuple-level granularity is optimal. We feel that this is unlikely as typically the value of n * m will be in the millions. Therefore for typical queries (unless there are millions of processors), tuple-level granularity places an unnecessary burden on the arbitration network without an apparent increase in performance. By sending pages of relations to the processors, a similar degree of concurrency can be achieved while minimizing network traffic.

The second problem caused by tuple-level granularity ap-

proach is that of memory management. An efficient mechanism must be provided to facilitate data transfers between the mass storage units and the cache memory. Any such mechanism relies on block transfers of data. Thus a relation in a database must be divided into a number of fixed size data pages. When looking for a candidate page in the cache to be thrown out the memory manager now must know whether additional tuples are expected to be placed in a page or not. One possible solution is to provide a second cache for tuples which are produced without a page to place them in. Although plausible, this considerably complicates the function of the memory manager.

4.0 A Preliminary Data-flow Database Machine

Based on our experience in the areas of database machine design and query processing in MIMD organizations, we have identified the following requirements for data-flow database machines. We feel each is very important for database machines in general and data-flow database machines in particular:

1) Concurrency Control. We feel that a database machine, in addition to the execution of a single query tree by several processors, must be able to support the simultaneous execution of multiple queries from several users if system resources are to be effectively utilized. This requires careful control of which queries are permitted to execute concurrently.

2) Distributed instruction initiation and control. In [6,10], the arbitration and distribution networks are responsible

for instruction initiation and data distribution. The arbitration network, in particular, provides a mechanism for initiating several enabled instructions simultaneously. For data-flow database machines, these networks are too general purpose. Instead the functions of each can be distributed thereby simplifying the complexities of these networks. This can be done without degrading performance since packets originating from one cell are sent to a fixed subset of the processors.

3) Flexible processor assignment arrangement. In order to optimally utilize the processors available, it must be possible to assign either one or all of the processors to an instruction.

4) Broadcast facility for join operations. When more than one processor is used to executed the "nested-loops" join algorithm, each processor will join a distinct set of pages from the outer relation with all the pages of the inner relation. In order to minimize data movement, a broadcast facility is needed so that a page from the inner relation can be distributed to some or all of the participating processors simultaneously. DIRECT [1,2] utilizes a cross-point switch to achieve this facility.

5) Expandability and Reliability. The database machine design should permit the addition of additional processors in a simple and straightforward manner and should be able to survive an arbitrary number of disabled processors.

4.1 Hardware Organization and General Operation

After careful consideration of the requirements for a data-flow database machine which were discussed above, we have developed a ring-based organization which is shown in Figure 4.1 below. This organization contains six major components:

1) The master controller (MC).
2) A set of instruction controllers (IC).
3) A communications ring (inner ring) connecting the master controller with the instruction controllers.
4) A mass storage system with a multiport disk cache.
5) A set of instruction processors (IP).
6) A communications ring (outer ring) connecting the instruction processors with the instruction controllers.

The MC serves a number of functions. The first is to handle communications with the host processor. When a user's query (in the form of a query tree) is received by the MC it is placed in a queue of queries awaiting execution. When system resources (ICs and IPs) become available, the MC removes the next query from the queue, checks it for concurrency conflicts with other executing queries, and then distributes a subset of the instructions from the query to a set of instruction controllers. The other functions of the MC are to control utilization of the disk cache among the ICs and to control instruction processor allocation.

The set of instruction controllers forms a distributed arbitration network. Each IC is generally responsible for control-

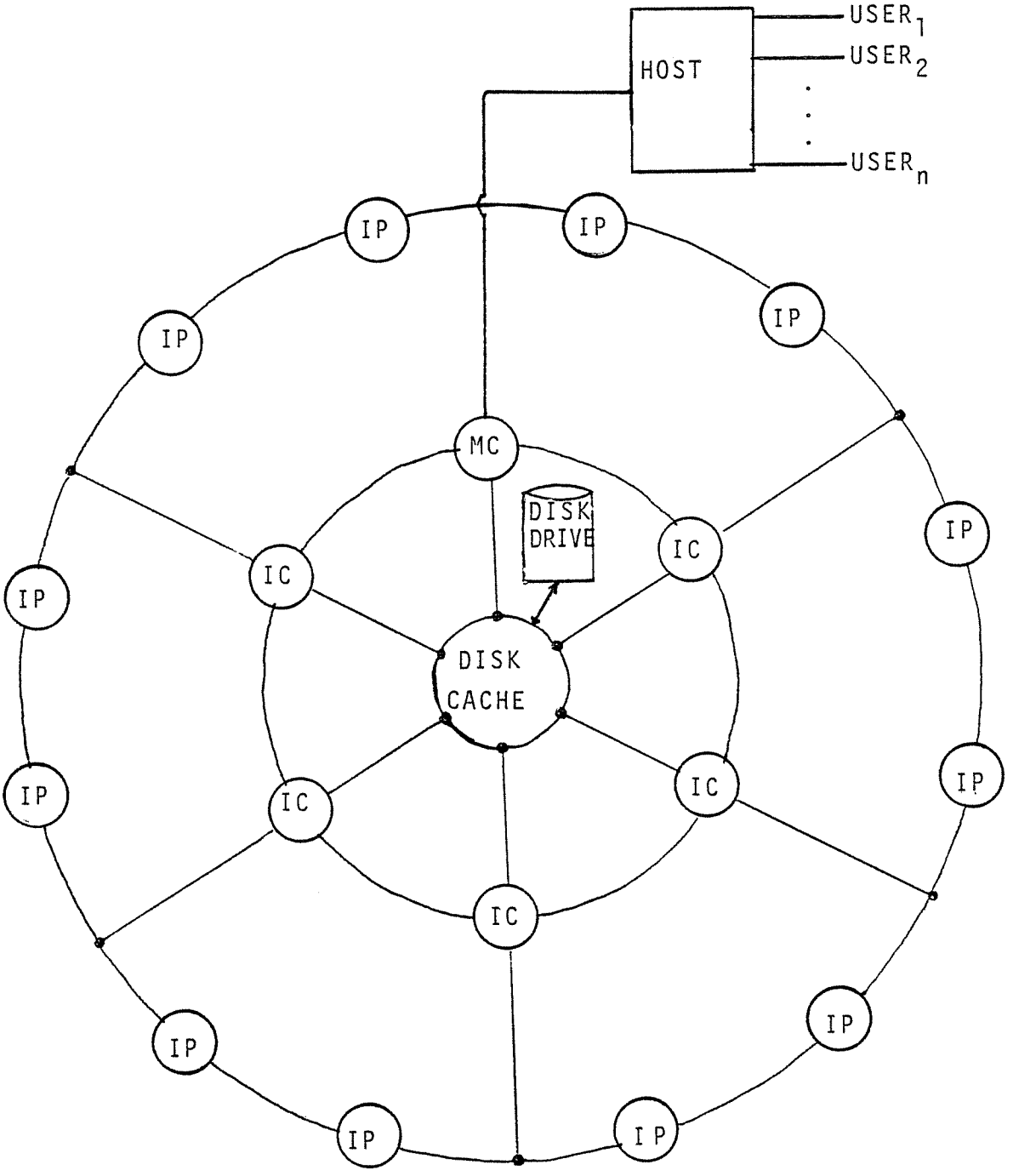


FIGURE 4.1

A DATA-FLOW DATABASE MACHINE CONFIGURATION

ling the execution of one instruction (e.g. a restrict, join, etc.) from a query. However, the situation may occur when a IC is controlling the execution of one instruction while receiving data for another instruction.

Controlling an instruction involves first acquiring a set of instruction processors from the MC and then distributing instruction packets (see Section 4.2) to the IPs which are allocated to the instruction by the MC. Thus the ICs compete with each other for the processors in the IP pool. The MC is responsible for arbitration of the requests in a manner which maximizes system performance by insuring that processors are distributed across all nodes in the query tree.

Each IC has a local memory for pages of source relations which will be used as operands in the instruction packets it distributes to the IPs. When the local memory of an IC fills, the IC will write the least desirable pages to its segment of the multiport disk cache. One possible approach for controlling usage of the disk cache is to divide it among the ICs according to the number of IPs each is controlling. When an IC fills its segment of the disk cache, pages will be swapped out to disk. Thus, the IC local memory, the disk cache, and the mass storage devices form a three-level storage hierarchy.

IPs are responsible for executing instruction packets which are placed on the outer ring by the ICs. When an IP receives an instruction packet addressed to it, it performs the operation specified in the packet and then produces an output packet. The

IP then places the output packet on the outer ring and sends it to the IC which is responsible for controlling the subsequent operation in the query tree. Thus, the IPs and the outer ring form a distributed distribution network for result packets.

The inner ring, as has been discussed above, is used exclusively for distribution of instructions and other other control messages by the MC. Since the messages required for such activities are small and limited in number, a bandwidth of 1-2 million bits per second (Mbps) should be sufficient.

The outer ring, on the other hand, is used for distribution of instructions and result packets by the ICs and IPs. Figure 4.2 represents the bandwidth requirements of DIRECT [4] with page-level granularity for the test data described in section 3.2. The following assumptions were made:

- 16K byte operands for instruction packets
- PDP LSI-11s as IPs (can read a 16K byte page in 33ms)
- The data cache is constructed from Intel 2314 CCD chips
- Two IBM 3330 disk drives for mass storage of relations
- A cross-bar switch with broadcast capabilities is used to connect the IPs with the data cache.

The bandwidth for each of the different processor levels was obtained by dividing the total number of bytes transferred by the execution time of the benchmark containing ten queries. Thus, the bandwidth values represent average values and not peak load values.

The ring organization which we propose to employ is that

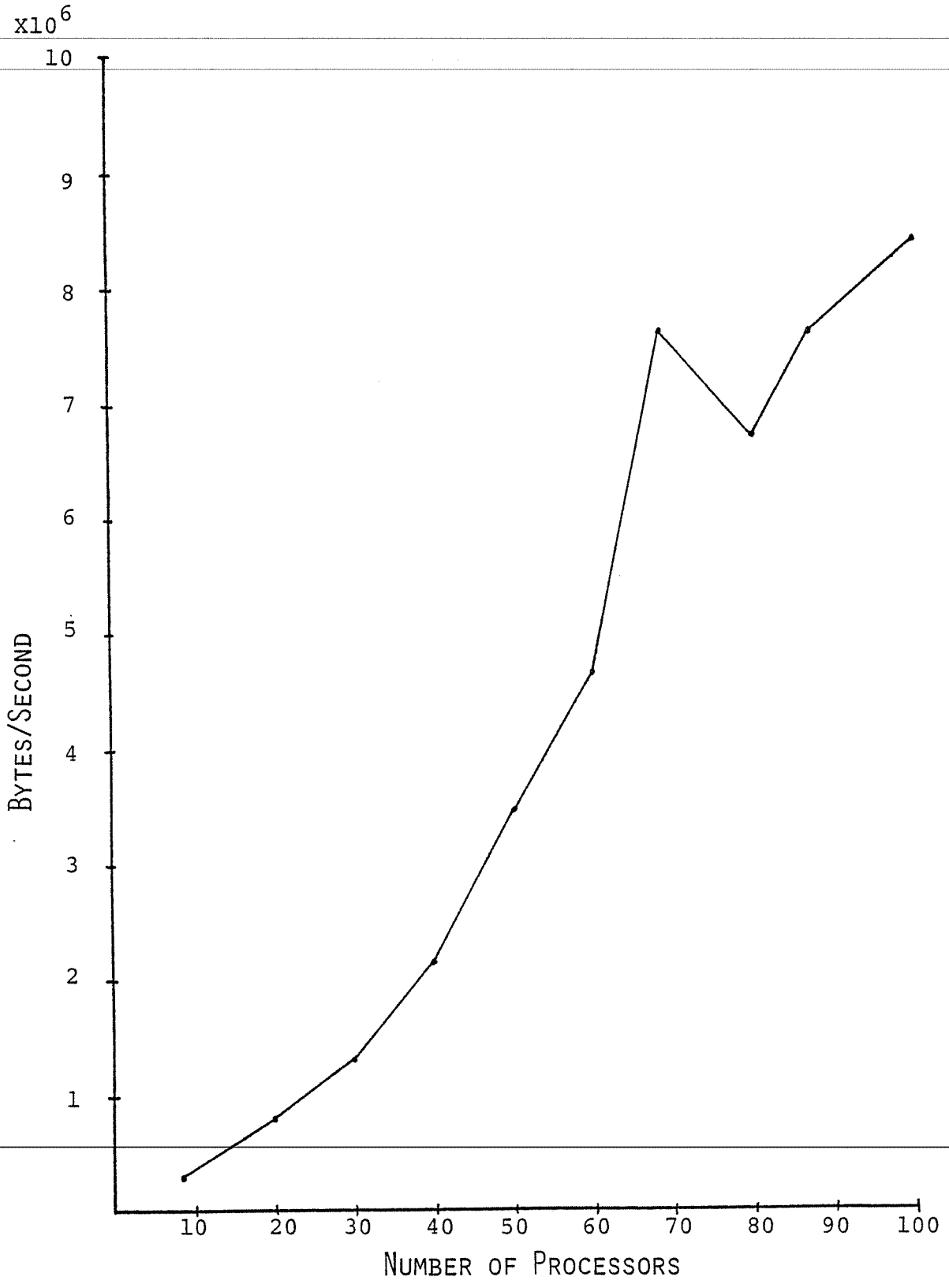


FIGURE 4.2

BANDWIDTH REQUIREMENTS OF DIRECT

which is used in the Distributed Loop Computer Network [13]. This network employs a technique known as shift-register insertion and can handle the transmission of variable length messages. It has been demonstrated [14] superior in several aspects to Newhall [15] and Pierce [16] loops. If 25 ns shift registers are used (AM25LS164 and 299), a ring bandwidth of 40 Mbps can be obtained. As indicated by Figure 4.2, this is sufficient for up to 50 instruction processors. For larger configurations requiring bandwidths of up to 100 Mbps there appear to be two alternatives. One expensive option is to use ECL shift registers which can be shifted at the rate of 1 bit per nanosecond for the loop medium. A more appealing alternative are loops constructed using fiber optic technology. Fiber optics can support bandwidths of 400 Mbps [17] and should be commercially available in the next 5-10 years.

4.2 Instruction Control and Execution

When an instruction is assigned to an IC it can be in one of two states. If the instruction's operand(s) are source relations in the database, then the instruction is ready to be executed. In this case the MC will also send to the IC a page table describing each operand. Otherwise, if the instruction is not enabled, the IC will first create a page table for each operand of the instruction and then wait for pages of the source operand(s) to arrive from IPs being controlled by another IC. As pages (which may not be full) arrive, they are compressed to form full pages

[3] and then stored in the IC's local memory or its segment of the disk cache.

When an IC is ready to initiate the execution of an instruction (i.e. at least one page of each operand is present), the IC first sends a control packet to the MC which requests an initial allocation of IPs and disk cache page frames. If the requested allocation cannot be fully satisfied, the MC will respond with a list of the IPs and page frames which are currently available. When another instruction has terminated, the MC will send the remaining requested resources to the IC.

After the IC has acquired a set of IPs and disk cache page frames it is ready to initiate the instruction by sending instruction packets to the IPs it is controlling. The packet format which is employed is shown in Figure 4.3. The destination of the packet is controlled by the IPid field. Upon receipt the IP applies the operation code to the data pages contained in the packet. Tuples of the result relation are first placed by the IP in an internal buffer. After the packet has been executed, the next step taken by the IP is controlled by the "flush-when-done" flag in the instruction packet. If the value of this flag is "yes", the IP places the result tuples in a result packet (Figure 4.4) and sends the packet to the IC specified by the "ICid of destination" field of the instruction packet via the outer ring.

After the result packet is sent or when the value of the "flush-when-done" flag is "no", the IP sends a control packet (Figure 4.5) to the IC which sent the instruction packet. This

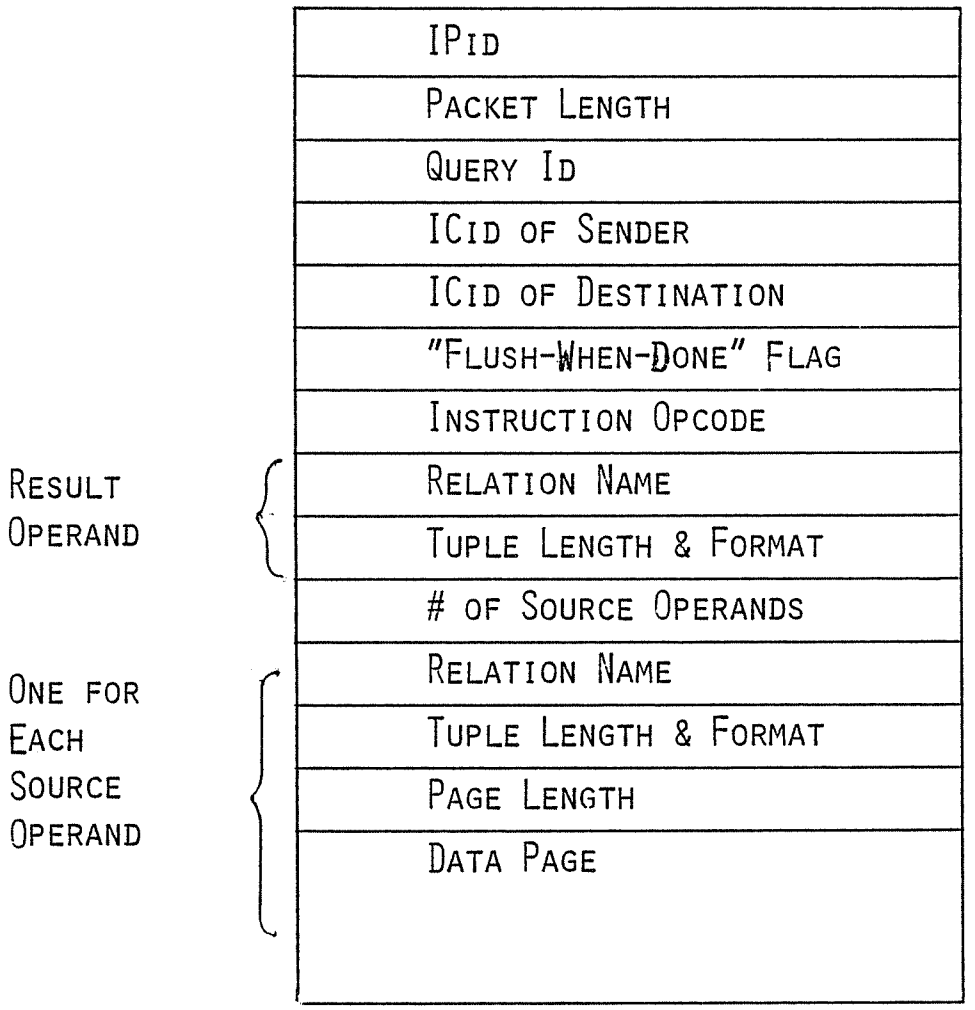


FIGURE 4.3

INSTRUCTION PACKET FORMAT

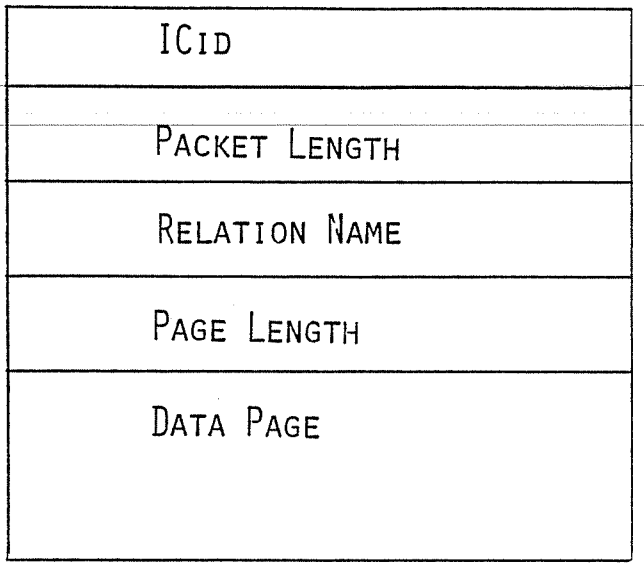


FIGURE 4.4

RESULT PACKET FORMAT

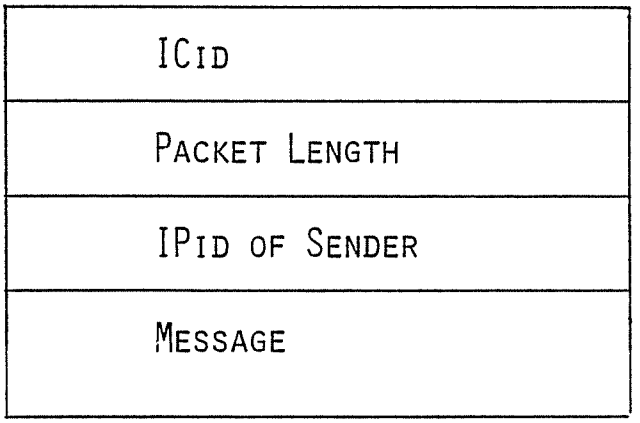


FIGURE 4.5

CONTROL PACKET FORMAT

is an indication that the IP has finished the task assigned and is ready for further work.

If the instruction being performed is a restrict operation, the IC can respond to this "done" signal, by sending the IP another data page from the source relation. If this is the last instruction packet to be sent to the IP, then the IC will set the "flush-when-done" field to yes. Then, when the IP sends the next "done" control packet, the IC can release the IP by sending a control packet to the MC releasing the IP.

When an IP first receives an instruction packet for a join operation, it sets up an "inner-relation control" (IRC) vector with one entry for each page of the inner relation. (Initially this vector will have only one entry, but the vector will grow as execution of the instruction progresses.) After the IP has joined the first page of the outer relation with the first page of the inner relation (the two operands in the packet), the IP will send a "done" control packet to the controlling IC. Included in this packet will be a request for the second page of the inner relation. The IC responds to this request by broadcasting the requested page to all IPs which are executing the join. (An IP can determine if a broadcast packet is meant for it by examining the Query ID field of the packet). Subsequent requests for the same page which are received by the IC "soon" afterwards can be ignored.

Each IP which receives the broadcast packet can be in one of several states. If the IP has already sent or is about to send a

request for the same page to the IC, then the IP can proceed to join the new page of the inner relation with its current page of the outer and update its IRC vector appropriately. If the IP does not have room in its local memory for the broadcast page, it will ignore the packet. However, the following scenario may occur. Because its local buffer is full, an IP ignores page i of the inner relation. When broadcast page i+1 is received (before it or page i has been solicited by the IP), the IP will read page i+1 and use it as an operand page. This situation can continue until a packet is received which indicates that this is the last page of the inner relation. At this point each IP will examine its IRC vector and then proceed to request those pages which it missed. When the IP has joined the current page of the outer relation with all the pages of the inner relation, it will first zero its IRC vector and then signal the IC that it is ready for another page of the outer relation which has not yet been distributed to an IP. In this way message traffic on the outer ring is minimized and yet correct operation of the join can be guaranteed.

5.0 Conclusions and Future Research

In this paper we have discussed alternative operand granularities for data-flow database machines and have demonstrated that page-level granularity is the best choice for optimum system performance. We have also presented a preliminary design for a data-flow database machine which utilizes page-level granularity

and supports distributed control of instruction execution.

There are several features of our proposed design with which we are not completely satisfied. In particular, we feel that it should be possible to route some of the data pages which are produced by IPs directly from one IP to another without first sending the page to an IC. If such an approach could be successfully implemented then message traffic on the outer ring could be further reduced. There appears, however, to be a tradeoff between decreased message traffic and increased IP complexity which needs further examination before the correct approach can be chosen.

Two other areas which need additional research are algorithms for performing the project operator (elimination of unwanted attributes and duplicate tuples) using multiple processors, and concurrency control. We have been examining the problem of the project operator for several months and have not yet developed an algorithm for which a high degree of parallelism can be maintained for the duration of the operator. We are also finalizing the design of a concurrency control mechanism for DIRECT and need to examine the effect of distributed instruction control on it.

With regard to equipment, we are in the process of acquiring ten PDP LSI-11/23 processors with NSF equipment grant MCS79-07516. These ten processors along with the five LSI-11s and the multiport CCD memory currently being used for DIRECT should give us a basis for a future implementation of a data-flow database

machine if suitable ring network hardware can be obtained.

6.0 Acknowledgements

Kevin Wilkinson's diligence and patience in listening and commenting on ideas is much appreciated.

REFERENCES

1. DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," Proceedings of the 5th Annual Symposium on Computer Architecture, April 1978, pp. 182-189.

2. DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June 1979, pp. 395-406. Also: Computer Sciences Technical Report #325, Univ. of Wisconsin, June 1978.

3. DeWitt, D.J., "Query Execution in DIRECT", Proceedings of the ACM-SIGMOD 1979 International Conference of Management of Data, May 1979, pp 13-22.

4. Boral, H., and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," submitted to ACM Transactions on Database Systems. Also Computer Sciences Technical Report No. 368, University of Wisconsin, October 1979.

5. Blasgen, M.W., and K.P. Eswaran, "Storage and Access in Relational Data Bases," IBM System Journal Vol. 16, No. 4, 1977, pp. 363-378.

6. Dennis, J.B., and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor,"The 2nd Annual Symposium on Computer Architecture: Conference Proceedings, January 1975, pp.126-132.

7. Arvind, and K.P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77: Proceedings of IFIP Congress 77, (B. Gilchrist, Ed.), August 1977, pp 849-853.

8. Davis, A.L., "The Architecture of DDM1: A Recursively Structured Data Driven Machine," Proceedings of the 5th Annual

Symposium on Computer Architecture, April 1978, pp. 210-215.

9. Rumbaugh, J.E., "A Data Flow Multiprocessor," IEEE Transactions on Computers, February 1977, pp. 138-146.

10. Dennis, J.B., and K.S. Weng, "An Abstract Implementation for Concurrent Computation with Streams," Proceedings of the 1979 International Conference on Parallel Processing, August 1979, pp. 35-45.

11. Smith, J.M., and P. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," CACM Vol. 18, No. 10, October 1975, pp. 568-579.

12. Yao, S. Bing, "Optimization of Query Evaluation Algorithms," ACM Transactions on Database Systems, Vol. 4, No. 2, June 1979, pp. 133-155.

13. Liu, M.T., and C.C. Reames, "A Loop Network for the Simultaneous Transmission of Variable Length Messages," Proceedings of the 2nd Annual Conference on Computer Architecture, January 1975, pp. 7-12.

14. Reames, C.C. and M.T. Liu, "Design and Simulation of the Distributed Loop Computer Network," Proceedings of the 3rd Annual Conference on Computer Architecture, 1976, pp. 124-129.

15. Farmer, W.D. and E.E. Newhall, "An Experimental Distributed Switching System to Handle Bursty Computer Traffic," Proceedings of the ACM Symposium on Data Communications, Pine Mountain, GA, October 1969, pp. 1-33.

16. Pierce, J.R, "Network for Block Switching of Data," Bell System Technical Journal, Vol. 51, No. 3, July/August 1972, pp. 1133-1143.

17. Frazer, W.D., "Potential Technology Implications for Computers and Telecommunications in the 1980s," IBM Systems Journal, Vol. 18, No. 2, 1979, pp. 333-347.