DATA-FLOW DATABASE MACHINES

by

Haran Boral
David J. DeWitt

Computer Sciences Technical Report #429

April 1981

Data-Flow Database Machines

Haran Boral
David J. DeWitt

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

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 #DAAG29-80-C-0041.

ABSTRACT

This paper presents a discussion of the application of data-flow machine concepts to the design and implementation of database machines that execute relational algebra queries. We begin with a description of the types of problems that are encountered in designing machines to efficiently process very large amounts of data. Next, the concept of data-flow query processing is introduced and we present results on the application of these techniques to query processing in relational database machines. 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. Introduction

For the past few years a number of investigations of data-flow languages and architectures have been undertaken [ACKE79a, ACKE79b, ARVI77, ARVI78, DAVI78, DENN75a, DENN75b, DENN80, PLAS76]. Most, if not all of these, can be characterized as dealing with pure data-flow. That is, the ultimate goal of these projects is to produce a general purpose data-flow machine. In this paper we describe a different line of research: the application of data-flow machine principles as a solution to a specific problem.

For the past several years our research efforts have concentrated on the problems associated with providing efficient access to relational databases that are too large to be handled by a single conventional processor [BORA80a, BORA80b, BORA81a, DEWI79a, DEWI79b, DEWI81]. In [DEWI79a] the design of DIRECT, a multi-processor multiple instruction stream multiple data stream (MIMD) relational database machine, is presented. A prototype of this machine which supports the relational database system INGRES [STON76] became operational in June 1980 using a PDP 11/40 and eight PDP 11/23s.

Until DIRECT was proposed, all previously proposed database were single instruction stream multiple data stream (SIMD) machines and thus could only execution one database operation at a time. One consequence of an SIMD design database machine is that there is no need to schedule the activities of the processors. With an MIMD design such as DIRECT it is possible to have groups of processors working on different instructions from the

same query, from different queries, or both. Therefore, after the design of DIRECT was completed we began to explore alternative processor allocation strategies for MIMD database machines. The goal of this research was to determine what strategy would maximize overall system performance in terms of the number of queries executed per second.

The results of this research on processor allocation strategies are presented in [BORA81a]. In this paper we will summarize the results presented in [BORA81a] and will describe our current efforts to design a new database machine that employs data-flow principles.

We begin in Section 2, with a discussion of relational database systems and the types of operations database machines for relational systems must support. In Section 3, we introduce the concept of dataflow query processing. We also present some of our early results on the application of these techniques for query processing in database machines for relational database management systems (DBMS). We outline the hardware organization and mode of operation of the proposed architecture in Section 4. In Section 5 we present some conclusions and future research plans.

2. Relational Database Systems

In this section we provide background information about the type and complexity of the operations that a multi-processor architecture for a relational DBMS must support.

2.1. Basic Concepts of Relational DBMS

A relational database [CODD70] consists of a number of normalized relations. Each relation is characterized by a fixed number of attributes and contains an arbitrary number of unique tuples. Thus, a relation can be viewed as a two dimensional table in which the attributes are the columns and the tuples are the rows. In a relational DBMS, relations are used to describe both entities and relationships between entities. For example, a simple database that describes information about suppliers and parts might contain three relations: S, P, and SP. The attributes of S would be those characteristics that are necessary to represent a supplier. Possible attributes might be supplier_#, supplier_name, and address. Each tuple in S would describe one supplier. The attributes of the P relation might be part_#, part_name, and part_weight. Thus S and P both describe entities. The SP relation would be used to associate suppliers with the parts they supply. Its attributes would be supplier_# and part_# (assuming each of these characteristics is capable of identifying a unique supplier in relation S and a unique part in relation P).

2.2. Operations on Relational DBMS

Access to a relational database is generally through a high-level non-procedural language that is based on either relational algebra or relational calculus [CODD70]. One example is the relational algebra language QUEL that is supported by INGRES [STON76].

The operations supported by QUEL and most relational DBMSs

can be divided into three classes according to the time complexity of the algorithms used on a uni-processor system. The first class includes those operations that reference a single relation and that require linear time (i.e. they can be processed in a single pass over the relation). The most familiar example is the selection operation which selects those tuples from a relation that satisfy a certain qualification (e.g. suppliers in N.Y.). This class also includes the scalar aggregates operations. Scalar aggregates are operations, such as average and sum, which are applied to one attribute of a relation.

The second class of operations are those that also operate on a single relation but require non-linear time O(nlogn). Included in this class of operations are projection and aggregate functions. Projecting a relation involves first eliminating one or more attributes (columns) of the relation and then eliminating any duplicate tuples that may have been introduced by the first step. Sorting (which requires O(nlogn) time) is the generally accepted way of eliminating the duplicate tuples. Aggregate functions are scalar aggregates applied to non-overlapping partitions of a single relation.

The final class of algorithms that must be supported are those involving two (or more) relations. Algorithms in this class also require non-linear time. The most frequently used operation from this class is the join. A join would be used by a user to find the name and address of all suppliers who supply part number 3. To perform this query, the user would first select those tuples from the SP relation with part_# = 3. Next,

the selected SP tuples would be joined with the tuples in the S relation on the supplier_# attribute of both relations. Finally, the supplier_name and address attribute values from the "joined" relation would be extracted. In effect, the join operation is a restricted cross-product. Another member in this class is the division operation.

Update operations are also supported by QUEL. They are not included in the above classification because QUEL allows the user to identify the tuples to be updated using a qualification clause of arbitrary complexity. Thus, the complexity of an update operation depends on the complexity of the qualification clause which is made up of the above described operations.

Because the size of a typical relation referenced in a query may be 10 million bytes (100,000 tuples each 100 bytes long) all algorithms for relational DBMS must be external algorithms. One very important consequence of this fact is that in any architecture executing operations on relations the number of I/O and inter-processor data exchanges must be minimized.

2.3. Parallel Algorithms for Relational Operations

In order to exploit the parallelism present in a multiprocessor database machine, parallel algorithms must be developed for each of the relational algebra operations that are supported. In this section, we present two parallel algorithms for the join operation. Several parallel algorithms for each of the remain-

ing[1] operations may be found in [BORA80b].

2.3.1. Parallel Nested-Loops Join Algorithm

The nested-loops join algorithm works by comparing each unit in one (the outer) relation with each unit in the other (the inner) relation. On a single processor a unit will generally correspond to a single tuple. We have identified two versions of this algorithm for multi-processor systems.

If the outer relation is N units long, the inner relation is M units long (M<N), and there are N processors available, then each processor can join one unit of the outer relation with the entire inner relation. N unit movements are required to distribute the units of the outer relation. After the outer relation has been distributed, the units of the inner relation can be read, one at a time, from mass storage and broadcast to all participating processors. Because the inner relation units are distributed to the processors via a broadcast mechanism, only O(N+M) unit I/O and intra-machine transfers are required to execute it. We therefore designate this algorithm the "N+M" algorithm.

In the second parallel nested-loops algorithm, which we term the "N*M" algorithm, pairs of units, one from each relation, are distributed to the processors. One can identify two possible advantages of this algorithm. First, by always distributing a

---

[1] Previous research [HAWT80, DEWI81] has shown that those operations requiring linear time on a single processor (e.g. selection) are most efficiently processed by a computer architecture which processes the operations "on the fly" as the data is read off the disk [SLOT70]. Therefore, only operations in the latter two classes are of interest in this paper.

unit from both relations in each packet there is no need to keep the processors synchronized as in the N+M algorithm. A second possible advantage is that the potential parallelism can be increased beyond N (the limit of the first algorithm). It should be noted that in order to achieve the high degree of parallel asynchronous activity O(N*M) intra-machine unit transfers are required.

2.3.2. Other Parallel Join Algorithms

Recent research has shown how sorting [BORA80b] and hashing [GOOD80] algorithms can be extended for use in a multi-processor environment. Both sorting and hashing form the basis of some efficient uni-processor join algorithms [BLAG77]. In [BORA80a], we demonstrate that a parallel version of the N+M nested-loops algorithm performs better than a join algorithm that uses parallel sorting.[2] Furthermore, controlling the execution of a parallel algorithm that employs either of the two nested-loops algorithms appears to be much simpler than controlling a multi-processor sort.

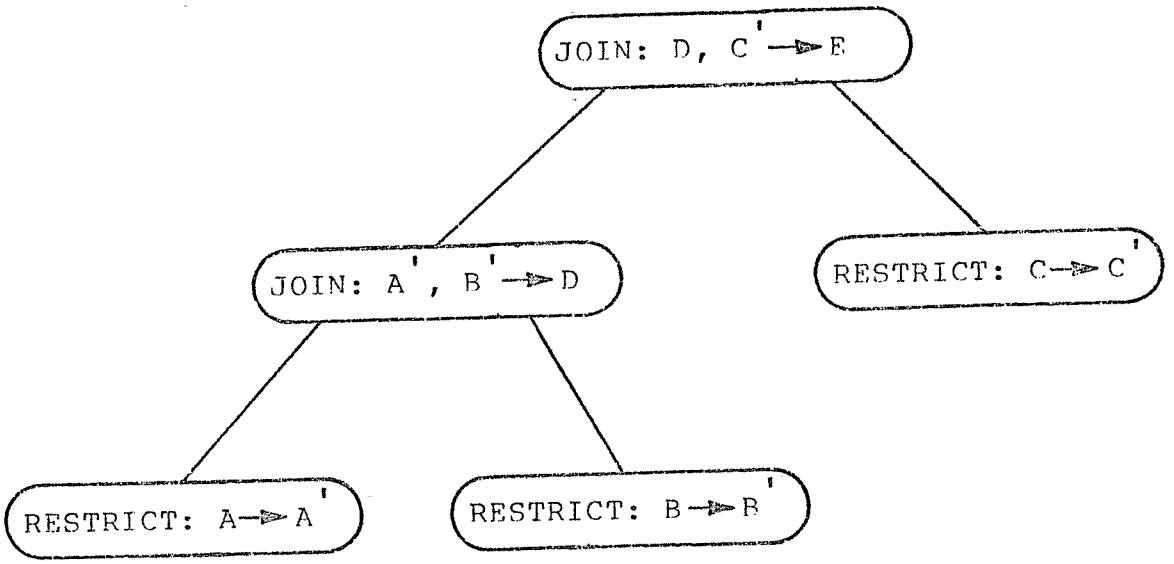
3. Data-Flow Query Processing

3.1. Relational Algebra Q ery Trees

Each relational algebra query is generally comprised of one or more operators and is organized in the form of a tree. Nodes

[2] Except for the case that the number of processor available for execution is much smaller than the number of units (fixed size pages were used) in either relation.

in such a query tree correspond to the relational algebra operators such as select, join, project or delete. Each node in the tree operates on one or more input relations and produces a single output relation. Nodes higher up in a query tree operate on relations produced by nodes immediately below them while the leaf nodes operate on permanent relations of the database. The output of any node is a temporary relation which is deleted at the end of the query execution. If the user wishes, he can make the result of a query a permanent relation in the database. An example of a relational algebra query in the form of a query tree is shown in Figure 1.



A Sample Query Tree
Figure 1

3.2. Operand Granularity for Query Execution

There are three possible operand granularities that can be used by a processor executing a relational algebra operation: an

entire relation, a page from a relation, or a tuple from a relation. For a conventional DBMS on a single processor, relation-level granularity is the obvious choice since little or no parallelism can be obtained. On the other hand, this level of granularity would obviously be a poor choice for a multi-processor system since it would severely constrain the potential parallelism.[3]

Tuple-level granularity also seems to be an unreasonable choice. First, a tuple will almost never be large enough to constitute an efficient unit of transfer between mass storage and a processor. Second, the amount of time required by a processor to process a single tuple will be shorter (possibly an order of magnitude less) than the time it would take to transfer that tuple between two points. Therefore, the processors in a machine using this granularity will most likely be under-utilized and the communication device overloaded.

Consequently, a page-level granularity that uses fixed-size pages seems to be the most attractive choice. The page size should be chosen such that: the page constitutes an efficient unit of transfer and the "average" relation would consist of a large enough number of pages to allow a high degree of parallel processing. It should be noted that transferring a page consisting of n tuples between two points will require less than n times the time to transfer a tuple since the overhead cost (which can be significant) will be amortized over the n tuples. Therefore,

---

[3] Unless, perhaps, if the processors are used in a pipelined fashion as suggested by [YAO79].

we expect that the time to process a page will be approximately the same as the time to transfer it between two points. This would lead to a balanced load on the processors and communication device resulting in better resource utilization.

3.3. Operand Granularity for the Scheduler

If page-level granularity is chosen as the unit of execution by the processors, there are two possible granularities that can be used by a scheduler to enable an instruction for execution: a relation and a page. When a relation-level granularity is used for task scheduling, then an operation is enabled for execution only after all of its input relations have been completely computed. If a page-level granularity is employeed for scheduling decisions, then an operation is enabled for execution as soon as one page of each input relation exists.

The relation-level scheduler must wait for all the operations producing the input relations to terminate before it can enable their parent operation to execute. Therefore, at the time that it enables an instruction the scheduler has complete information about the instruction: the number of pages in all operands, the average "fullness" of the pages in each relation, and their physical placement (in processors' memories, a cache, or mass storage). This information can be used by the scheduler in deciding how many processors to allocate for the execution of the operation, whether one of the relations should be compressed [DEWI79], and whether the operation should be enabled immediately or later. The disadvantage of this approach is that pages

that are generated by one operation may have to be stored (temporarily) in a cache or on a mass storage device until the parent (consuming) operation is enabled. This may pose severe problems because the high level of I/O traffic may slow down the execution of the consuming operation.

The page-level scheduler seems appealing precisely because it would not suffer from this problem. After an initial start-up time, processors would be distributed across most, if not all, of the nodes in the query tree. Pages would be pipelined from processors executing a child node to processors executing its parent soon after they were produced. Therefore, the number of I/O operations would be minimized because storage for temporary relation pages would not be required.

Our intuitive feelings were that page-level scheduling combined with the N*M version of the nested loops algorithms would yield the best performance. The N*M algorithm, since it corresponds to unfolding the two nested loops of the serial join algorithm, is similar to many of the algorithms suggested for use in data-flow machines such as matrix multiplication. These algorithms employ the maximal loop unfolding as a means for attaining a high level of execution concurrency. We believed that the high degree of parallelism, when coupled with the pipelining behavior of the scheduler, would lead to the most efficient use of the machine resources and yield the optimal performance level. In the following section we will discuss an experiment we conducted to evaluate the alternatives.

3.4. Comparison of the Schedulers

In this section we describe the results of a comparison of the two schedulers. These results were obtained from a detailed simulation model of DIRECT. We begin this section with an overview of the architecture of DIRECT in an attempt to place the problem of implementing the schedulers in a specific context. We then outline the results of our simulations without providing the numerical data. Including the numerical data in this paper would entail a more detailed understanding of the architecture of DIRECT and the experiments conducted than this paper could provide. The interested reader is referred to [BORA81a] for a more complete description of both the architecture and the experiments.

DIRECT is a centrally controlled MIMD multi-processor database machine. A DIRECT configuration consists of six component types:

(1) A number of processors termed query processors
(2) A number of CCD memory modules that serve as a disk cache
(3) A number of mass storage devices on which the database resides
(4) A cross point switch between the query processors and the modules (page frames) of the disk cache
(5) A controlling processor termed back-end controller interconnected to the query processors via a bus
(6) A general purpose computer through which the users communicate with the database machine.

All query processor intercommunication is through the shared cache. Management of the cache is a function of the back-end controller. Therefore, all reading and writing of the cache must be coordinated by the back-end controller through the use of

explicit message exchanges with the query processors. In order for a processor to write a page of a temporary relation it must first request a cache memory module address from the back-end controller. In the event that the cache is full the back-end will have to page some of its contents out to disk. Later, when a page that was written out to disk is needed it will first have to be brought back into the cache before the processor could read it.

We compared three different cases: the relation-level scheduler using the N+M nested loops algorithms which we term scheduler A, the page-level scheduler using the same algorithm which we term scheduler B, and the page-level scheduler using the N*M algorithms which we term scheduler C. Our simulation yielded a number of interesting results. First, the performance of scheduler B was approximately 35% better than schedulers A or C both of which demonstrated approximately the same level of performance. The performance of scheduler A, as anticipated, suffered because of excess thrashing between the cache and the mass storage device. Examination of the simulation results indicated that time after time, because a parent operation could not be initiated early enough some of the pages from one or both of its input relations had been paged out to disk.[4]

We were, however, very surprised about the relatively poor performance of scheduler C. When running scheduler C we found that processors were indeed distributed across many of the nodes

---

[4] Alternative page replacement algorithms for the cache were tested and the best one was used in all experiments.

of a query tree and that pages were being pipelined between them. Only a minimal amount of pipelining activity was displayed by scheduler B. Both schedulers B and C did significantly less disk I/O than did scheduler A. Closer examination of the results revealed that the poor performance of scheduler C was a consequence of the number of inter-processor communications required to distribute pairs of pages to the processors. As pointed out in Section 2.3.1, the number of interprocessor communications required by the M*N join algorithm utilized by scheduler C is quadratic in the size of the relations while for the other two schedulers (which used the N+M algorithm) this number was linear.

3.5. Implications for Data-flow Database Machines

Another potential problem which our research exposed is that the cost of controlling the activity of the query processors (measured in terms of the number of control messages) may exceed the cost of query execution. This effect was seen in all of the scheduling strategies evaluated[5] although both page-level schedulers required more control messages than the relation level scheduler. We believe that this is not a consequence of data-flow query processing but rather of the DIRECT architecture with its centralized controller. Thus, we feel that a properly designed data-flow database machine with decentralized instruction control should be able to minimize the impact of control messages on system through-put. We will present an overview of

[5] Our original study included other, non-data-flow, processor allocation strategies not mentioned here.

one possible design in the next section.

A second problem our research exposed was that the cost of distributing operand packets in the M*N join algorithm offset any gains in execution time due to the massive level of concurrency employed. This result is interesting because it seems to confirm similar results presented in [GOST80]. In this study the Irvine data-flow architecture was simulated and the performance of several algorithms of different types was evaluated. It was shown that the cost of communication between processors dominated the total execution time. We feel that these results imply that data-flow machines that use massive parallelism without due regard to the cost of communication among processors may not attain the expected performance level.

It appears that if communications costs increase linearly in the size of the data then execution time decreases as expected. However, if communications costs increase quadratically in the size of the data, then the expected level of performance will not be achieved. Thus, for data-flow database machines which must execute external algorithms due to the size of the relations referenced, the N+M nested loops algorithms in particular, and broadcast-based algorithms, in general, will be superior in performance. While these algorithms achieve only minimal pipelining, they do minimize I/O activity and inter-processor communication.

4. A Preliminary Architecture for a Data-flow Database Machine

Based on the results of the research described in the previous section and our experiences with simulating and implementing DIRECT, we established the following design objectives for our data-flow database machine:

(1) MIMD organization - A data-flow database machine must have a MIMD organization capable of supporting concurrent execution of operations from any number of user queries. This is necessary if the system is to provide good response as a back-end to a multi-user DBMS.

(2) Distributed instruction control - A data-flow database machine must support distributed control of instruction execution so that no one component will become a bottleneck.

(3) Broadcast algorithms - As presented in Section 3, our results indicate that algorithms with a linear number of data exchanges result in superior performance. In [BORA80b] broadcast-based algorithms for all the relational algebra operators are described.

(4) Minimal custom hardware - Our experiences with fabricating customized hardware (i.e. DIRECT's shared CCD cache) indicate that at least for a university environment customized hardware should be avoided as much as possible.

(5) Incrementally expansible - If our ideas are to ever become commercially viable, the architecture must be incrementally expansible so that a user can begin with a modest system and add additional resources as his needs grow.

In the following sections we will sketch the organization and operation of a data-flow database machine which we feel meets this set of design criteria. In Section 5, we will comment on its implementation.

4.1. Organization and General Operation

In this section we sketch the design and operation of a data-flow database machine. Our architecture consists of five main components:

(1) The master controller (MC).

(2) A set of instruction controllers (IC).

(3) A set of instruction processors (IP).

(4) A mass storage system

(5) A very high bandwidth communications medium with broadcast capabilities connecting the components of the system together.

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 function of the MC is to control IP allocation among the ICs.

Each IC is responsible for controlling one[6] instruction. Controlling an instruction involves first acquiring a set of IPs from the MC and then distributing instruction packets (see Section 4.4) to the allocated IPs. Thus the ICs compete with each other for the processors in the IP pool. (See [BORA81a] for a discussion of the number of IPs an IC should attempt to acquire for an instruction). 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 a query tree.

Each IC has a small[7] mass storage device in addition to its

---

[6] Possibly more than one.

local memory. The IC uses this two level memory hierarchy to store pages from the relations which are operands of the instruction it is controlling and which will be distributed in instruction packets. Each IC will attempt to keep the "most desirable" pages in its local memory.

IPs are responsible for executing instruction packets received from an IC on the communications network. 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 network and sends it to the IC which is responsible for controlling the subsequent operation in the query tree. Thus, the IPs and the network form a distributed distribution network [DENN75a] for result packets. We will discuss the interconnection network in more detail in Section 4.3 below.

4.2. Distributed Control of Instruction Execution

Although the architecture of some of the previously proposed data-flow machines might be usable as the basis of a data-flow database machine, we feel that our proposed design is much simpler and will be capable of achieving the same performance level. For example, consider the arbitration and distribution networks of the MIT machine [DENN75a]. These networks are responsible for instruction initiation and distribution of results. The design of the data distribution network is rela-

---

[7] 40 Mbyte winchester disks can be purchased today, with a controller, for approximately $6,000.

tively straightforward. Its function is to take a result packet produced by a processor and store it in those instruction cells which are specified in the packet header. The arbitration network, on the other hand, is very complex. It must continuously monitor all instruction cells and provide a mechanism for initiating several enabled instructions simultaneously by routing the contents of each enabled instruction to a free processor for execution. We feel that for data-flow database machines these two networks are too general-purpose and consequently excessively expensive.

In our approah we have replaced the instruction memory and the arbitration and distribution networks with a small number of relatively low-performance processors (the ICs). Each IC is responsible for controlling the execution of a few (perhaps only one) relational algebra operations. Thus, control of the execution of a query is distributed among a set of processors. If a typical query contains five operations, then fifty ICs can maintain a multiprogramming level of at least ten in the database machine.

Our approach appears to be viable for two reasons: program size (number of instructions) and execution time of a typical instruction. One frequently mentioned application for data-flow machines is large scientific programs (e.g. weather programs). These programs generally consist of thousands of instructions each of which takes only a few microseconds (or less) to execute. Even if the instruction operates on operands of type vector, multiple processors can be used to work on individual elements and

hence instruction execution time will still be in the microsecond range. For these applications a large instruction memory is required to hold the entire program. Since each instruction cell has one input to the arbitration network, the size of the arbitration network is proportional to that of the instruction memory. The arbitration and distribution networks must also be extremely fast. For example, if 100 one microsecond (execution time of a typical instruction) processing elements are to be kept busy, the arbitration network must be capable of routing 10^8 packets/second.

Relational algebra queries, on the other hand, are composed of relatively few instructions (typically 1-10 operations) each of which takes a relatively long time to execute (in the millisecond to second range). Also packets originating from one IC are sent to a fixed subset of instruction processors, as, for example, are the inner relation pages in the N+M nested loops join algorithm. This permits us to replace the instruction memory and the two networks with a set of processors without any loss of performance or functionality.

4.3. Interconnection Network

4.3.1. Technology

The communication hardware used to interconnect the MC, ICs, IPs, and mass storage devices must be able to support MIMD activity and communications of two types: point-to-point and broadcast. One medium which appears viable is based on a broadband, coaxial cable that uses frequency-multiplexed, RF-modulated

channels to allow for several simultaneous communications over a single piece of coax [MAGL80].

The transmission technology used is CATV which can support a frequency spectrum of between 300 and 400 Mhz. If a transmission bandwidth of between 2.5 and 3 Hz is required per bit, then one piece of coax can support a total communication bandwidth of approximately 100 Mbits/second. The total number of separate communications channels available is dependent on the bandwidth of each channel. If 1 Mbit channels are adequate then the communications network can support 100 channels. If 10 Mbit channels are required then only 10 channels can be supported and additional pieces of coax may be necessary. Our intuition (based somewhat on typical load levels of 3 Mbit ETHERNETs) is that 1 Mbit per channel will be adequate.

4.3.2. Network Utilization

The multiple channel capability will be used in two different ways. A single, specially designated, channel, which we term the control channel, would be used for coordinating activities on the machine. For example, processors that wish to establish a link among themselves (e.g. an IC and the IPs it is controlling) will obtain a reserved channel through the use of the control channel. Once a reserved channel has been assigned to them they can switch frequency and proceed with their "session", undisturbed, over their own reserved channel.

In our architecture the MC will be responsible for all resource assignment. It will always monitor the control channel.

In its idle state, any processor (IC or IP) will be listening to the control channel. At the time that the MC decides to initiate the execution of an instruction it picks an IC to control it and allocates some IPs and a channel to the IC. The processors switch frequencies to operate in the assigned channel (their operation will be described in the following section). At the time that a processor terminates its current task it switches frequency back to the control channel, informs the controller that it is ready for a new task and waits for a new assignment message.

4.4. Instruction Control and Execution

In this section we illustrate the operation of the machine in the execution of a single join operation. We call the reader's attention to the fact that in general a query tree will include more than one operation. In executing a query tree with more than one operation various other considerations must be taken into account. These are, however, beyond the scope of this paper.

When an instruction is assigned to an IC it can be in one of two states. If the instruction's operands exist (either they are source relations in the database or they have been computed) 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 (the input relations have not been computed) 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 other ICs. As pages (which may not be full) arrive, they are compressed to form full pages [DEWI79b] and then stored in the IC's local memory or in its disk.

When an IC is ready to initiate the execution of an instruction (i.e. at least one page of each operand is present), it 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.

Initially the IC broadcasts a copy of the code to be executed to the IPs. Also sent is the channel frequency to be used by the subsequent instruction's IC. After each IP receives a copy of the code to be executed it sets up an "inner-relation control" (IRC) vector with one entry for each page of the inner relation. Initially this vector will be empty. As the execution of the instruction progresses it will grow.

Execution of the algorithm begins with the distribution of the outer relation pages. Each IP receives one page. Next, the IC begins broadcasting the inner relation pages. When an inner relation page arrives, the IP marks it in its IRC and joins it with its outer relation page. Tuples of the result relation are placed in an internal buffer. Should the internal buffer fill, the IP will switch frequency to the subsequent instruction's channel and relay the result page to that instruction's IC. After emptying its buffer the IP returns to the original channel

where it continues execution.

After the last inner relation page has been broadcast the IC broadcasts an "end_of_inner_relation" message which indicates the number of inner relation pages actually broadcast. Each IP checks its IRC to see if it missed any pages and informs the IC of its state. After the IC has received rebroadcast requests from all of its IPs (there may be none) it begins the rebroadcast phase. When the IP finishes joining its current page of the outer relation with all the pages of the inner relation it informs the IC. In the event that additional outer relation pages exist these will be distributed to the IPs. Each IP will zero its IRC vector and the algorithm will be executed again. Otherwise the IP flushes its remaining output buffer contents and returns to the control channel informing the MC that it is idle.

5. Conclusions and Directions for Future Research

In this paper we have discussed the application of data-flow machine concepts as a means for enhancing the performance of a relational database machine. We used the results from a simulation study of DIRECT to show that a processor allocation strategy based on the data driven model of execution can indeed improve performance. We have also exposed two serious problems.

The first problem is that the cost of controlling the various processors in DIRECT dominated the execution time cost of the various queries we benchmarked. Although, the cost of controlling the processors was higher for the data-flow strategies than for the non-data-flow strategies we were able to conclude that

the primary reason for this behavior was the centralized nature of the architecture. We have therefore embarked on a new design of a database machine, based to an extent on DIRECT, which uses data-flow scheduling and decentralized control of instructions.

A preliminary design of this architecture was described in [BORA80a]. In Section 4 of this paper we outlined a more recent design. A more detailed description is in [BORA81b, BORA81c]. One of the attractive features of this new architecture is that it can be constructed using "off the shelf" components. The Computer Sciences Department at the University of Wisconsin has recently received a Coordinated Experimental Computer Research Program grant from NSF. Included in this grant are funds to be used for the purchase of 50 VAX processors to be interconnected using a broadband, multiple-frequency interconnection device. We intend to implement our architecture on this equipment during the next 2-3 years.

The second problem we exposed has, we believe, far reaching consequences. We showed that a data-flow scheduler that used a nested loops algorithm with maximal loop unfolding did not perform as well as expected because of the high level of interprocessor communication. The reason we believe this result to be of some consequence is that it is not isolated. Gostelow and Thomas [GOST80] arrived at a similar result under different architectural assumptions for a different set of problems. Additional study is needed to expose the tradeoffs between massive parallelism and the cost of communication between processors. A first step in this direction has been taken by Arvind [ARVI80].

This problem is also under study at the University of Wisconsin both for the relational database problem and for more general purpose programs.

Studying the tradeoffs between parallelism and communication costs may lead to a better understanding of both algorithms and architectures. This may result in the design of new algorithms (and consequently architectures) that possess properties that enable their execution using massive parallelism without incurring high penalties due to communications.

6. References

[ACKE79a] Ackerman, W.B. and J.B. Dennis, "VAL -- a Value Oriented Algorithmic Language: Preliminary Reference Manual," TR-218, Laboratory for Computer Science, MIT, 1979.

[ACKE79b] Ackerman, W.B., "Data-Flow Languages," Proc. of the 1979 NCC, Vol. 49, AFIPS Press, Montvale N.J., 1979

[ARVI77] Arvind and K.P. Gostelow, "A Computer Capable of Exchanging Processors for Time," Information Processing 77, B. Gilchrist, North Holland, N.Y., 1977.

[ARVI78] Arvind, Gostelow, K.P, and A W. Plouffe, "An Asynchronous Programming Language and Computing Machine," UC-Irvine Technical Report, December, 1978.

[ARVI80] Arvind, "Decomposing a Program for Multiple Processor Systems," Proc. of the 1980 International Conference on Parallel Processing.

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

[BORA80a] Boral, H. and D.J. DeWitt, "Design Considerations for Data-flow Database Machines," Proceedings of the 1980 SIGMOD International Conference on Management of Data, May 1980, Santa Monica, Calif.

[BORA80b] Boral, H., DeWitt, D. J., Friedland, D., and W. K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," submitted to ACM Transactions on Database Systems, October, 1980.

[BORA81a] Boral, H., and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," to appear ACM Transactions on Database Systems, June 1981.

[BORA81b] Boral, H., Ph.D. Dissertation, University of Wisconsin-Madison, May 1981.

[BORA81c] Boral, H., and D.J. DeWitt, In preparation.

[CODD70] Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," CACM Vol. 13, No. 6, June, 1970.

[DAVI78] Davis, A.L., "The Architecture of DDM1: A Recursively Structured Data Driven Machine," Proc. of the 5th Annual Symposium on Computer Architecture, April, 1978.

[DENN75a] Dennis, J.B. and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," Proc. of the 2nd

Annual Symposium on Computer Architecture, January, 1975.

[DENN75b] Dennis, J.B., "First Version of a Data Flow Procedure Language," MIT Memo LCS TM 61, May 1975.

[DENN80] Dennis, J.B., "Dataflow Supercomputers," IEEE Computer, Nov. 1980.

[DEWI79a] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers, June 1979, pp. 395-406.

[DEWI79b] DeWitt,D.J., "Query Execution in DIRECT," Proceedings of the 1979 SIGMOD International Conference on Management of Data, May 1979, Boston, Mass.

[DEWI81] DeWitt, D.J. and P. Hawthorn, "A Performance Evaluation of Database Machine Architectures," Invited Paper, to appear: Proceedings of the 1981 VLDB Conference, September, 1981.

[GOOD80] Goodman, J.R. Ph.D. Dissertation, University of California Berkeley, Nov. 1980.

[GOST80] Gostelow K.P. and R.E. Thomas, "Performance of a Simulated Dataflow Computer," IEEE Transactions on Computers, Oct. 1980.

[HAWT80] Hawthorn P. and D.J. DeWitt, "Performance Evaluation of Database Machines," To appear IEEE Transactions on Software Engineering.

[MAGL80] Maglaris, B. and T. Lissack, "An Integrated Broadband Local Network Architecture," Proceeding of the 5th Annual Local Computer Networks Conference, Oct. 1980.

[PLAS76] Plas, A. et. al., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," Proc. of the 1976 International Conference on Parallel Processing.

[SLOT70] Slotnik, D.L. "Logic per Track Devices" in "Advances in Computers", Vol. 10., Frantz Alt, Ed., Academic Press, New York, 1970, pp 291 - 296.

[STON76] Stonebraker, M. R. et. al., "The Design and Implementation of INGRES," TODS, Vol 1, No. 3, September 1976.

[YAO79] Yao, S. Bing, "Optimization of Query Evaluation Algorithms," ACM TODS Vol. 4, No. 2, June, 1979.