PROCESSOR ALLOCATION STRATEGIES FOR MULTIPROCESSOR DATABASE MACHINES

by

Haran Boral
David J. DeWitt

Computer Sciences Technical Report #368

October 1979

Processor Allocation Strategies for Multiprocessor 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 #DAAG29-75-C-0024.

ABSTRACT

In this paper we describe and evaluate four alternative strategies for assigning processors to queries in multiprocessor database machines. Our results demonstrate that SIMD database machines are indeed a poor design when their performance is compared with that of the three MIMD strategies which we present.

We also introduce the application of data-flow machine techniques to the processing of relational algebra queries. This strategy is shown to be superior to the other strategies described by several experiments. Furthermore, if the data-flow query processing strategy is employed, our results indicate that a two-level storage hierarchy (in which relations are paged between a shared data cache and mass storage) does not have a significant impact on performance.

1.0 Introduction

During the past several years we have been investigating the design and implementation of a database machine for the execution of relational algebra queries. The architecture of DIRECT can be found in [1,2]. In [3] we describe query organization and execution in DIRECT and introduce the problem of relation fragmentation and its impact on query execution time. In parallel with these efforts, we have been proceeding with the implementation of a prototype of DIRECT which we expect will be operational during the fall of 1979.

An important problem which arises in multiple-processor database machines is that of processor assignment. If the database machine serves as a back-end to a database system which simultaneously serves many users, there will generally be more than one query which is ready to be executed by the database machine. Given that each query consists of several relational algebra operations (e.g. restricts, projects, joins, or update operations), there are several possible strategies for assigning processors to these queries for execution. In the following sections we first will describe four alternative strategies and then compare and contrast each with respect to several performance criteria. The best of these strategies is shown to be a strategy which is based on the application of data-flow machine techniques to the execution of relational algebra queries. This technique for executing queries yields an average performance improvement

of 4:1 over the SIMD strategy and a 2:1 improvement over the other MIMD strategies. Furthermore, we also demonstrate that the impact of moving data pages between mass storage and a shared CCD cache is negligible for the data-flow query processing strategy.

The purpose of this paper is not, however, to compare the performance of DIRECT with a relational database management system on a conventional computer. Such a comparison has already been performed for RAP[4] and we believe that this paper will demonstrate that the performance of DIRECT is generally superior to that of "RAP-like" SIMD architectures.

Our investigation is restricted to retrieval operations. This was done for a number of reasons: (a) Update operations take a relatively short time to execute (compared to joins) and basically consist of a retrieval operation followed by modification of the selected tuples. (b) When considering update operations one must provide a concurrency control mechanism (in our case a number of such mechanisms). We feel that while these mechanisms would impact performance in a very real way they would serve to increase the complexity of the experiments and contribute considerably to hiding the salient features of the different strategies. We fully intend to consider the impact of update operations and concurrency control in the near future.

Section 2 contains an overview of DIRECT. In Section 3 four processor assignment strategies are introduced and described in detail. Section 4 describes the experiments which were conducted to compare the performance of each strategy. The results of

these experiments are presented in Section 5.

2.0 Overview of DIRECT

In this section the hardware and software characteristics of DIRECT will be reviewed. For more details the reader is encouraged to examine references [2] and [3].

2.1 Hardware

The hardware used to implement DIRECT consists of six main components: a host processor, a back-end controller, a set of processors for executing queries, a set of CCD memory modules which are used as pseudo-associative memories, an interconnection matrix between the set of processors and the set of CCD memory modules, and one or more mass storage devices. A diagram of these components and their interconnections can be found in Figure 2.1.

The host processor handles all communications with the users. A user who wishes to use DIRECT logs onto a modified version of INGRES [5] and proceeds in the normal manner. When the user executes a query, INGRES compiles the query into a "query packet" and sends it to the back-end for execution. Each "query packet" is comprised of one or more relational algebra operations (instructions) which are organized in the form of a tree. Each node corresponds to a relational algebra operation (actually update operations are decomposed into anumber of nodes) and is in a form which can be executed directly by an arbitrary number of

processors without interpretation. Figure 2.2 contains an example of a query tree.

The back-end controller is responsible for interacting with the host processor and controlling the processors. After it receives a query packet from the host, it determines the number of processors that should be assigned to execute the packet or an instruction within the packet, and then, as processors become available, assigns them to instructions.

Each processor is a PDP LSI-11/03 with 28K words memory. The function of each processor is to execute instructions assigned by the back-end controller.

Since DIRECT is a virtual memory database machine each relation in the database is divided into fixed size pages. Each CCD memory module corresponds to a page frame and can hold one page of a relation. The set of CCD memory modules acts as a disk cache for the mass storage devices and is used by the processors for relation storage during query execution. The cache is managed using a modified LRU algorithm.

The processors are connected to the CCD memory modules through an interconnection matrix [1,2] which has two properties important for supporting both intra and inter-query concurrency. The first property is that two or more processors, each executing the same or perhaps different queries, can simultaneously read the same CCD memory module. The second property is that two or more processors can simultaneously access different CCD memory modules (perhaps exclusively in the case of writing).

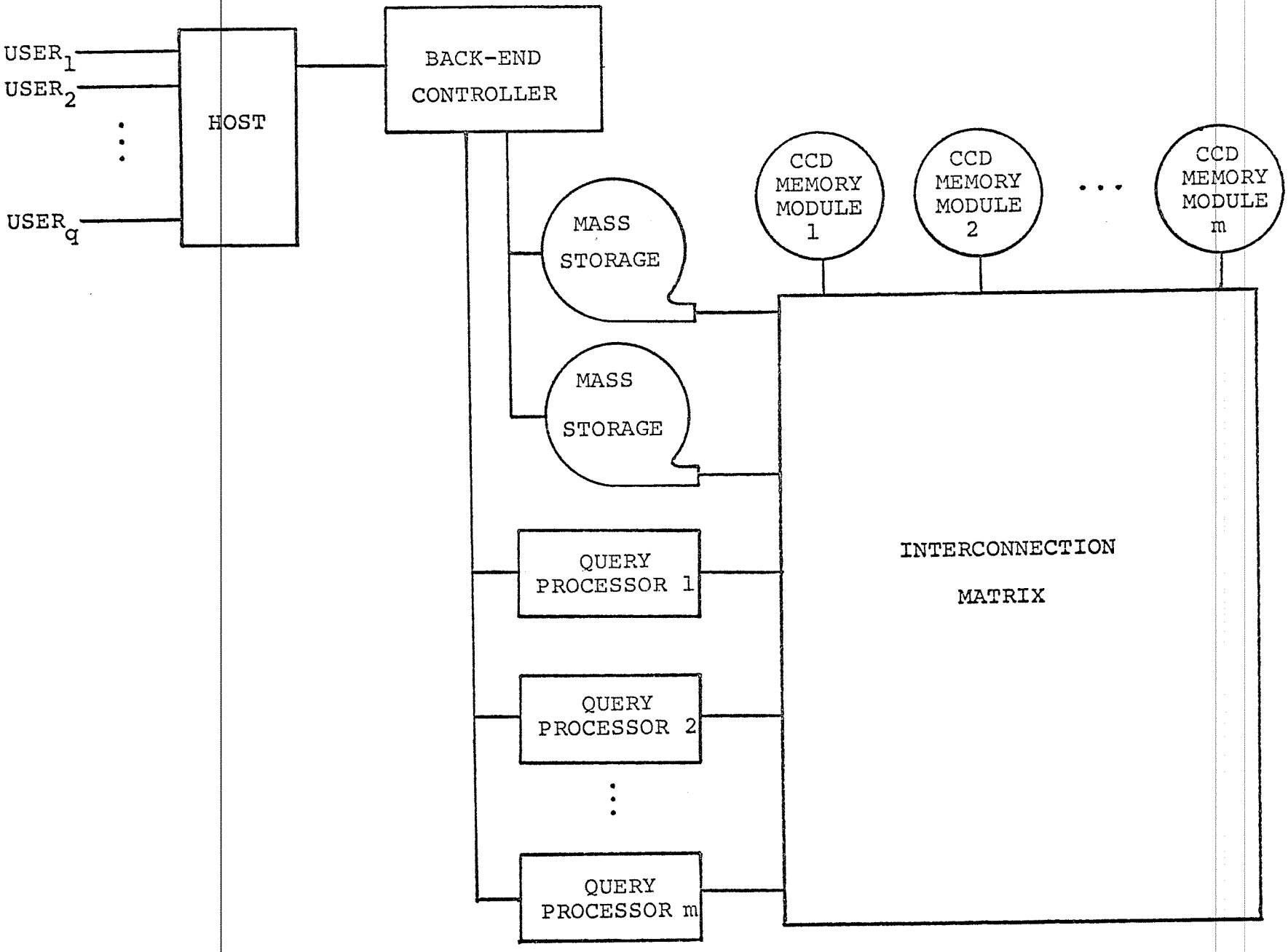
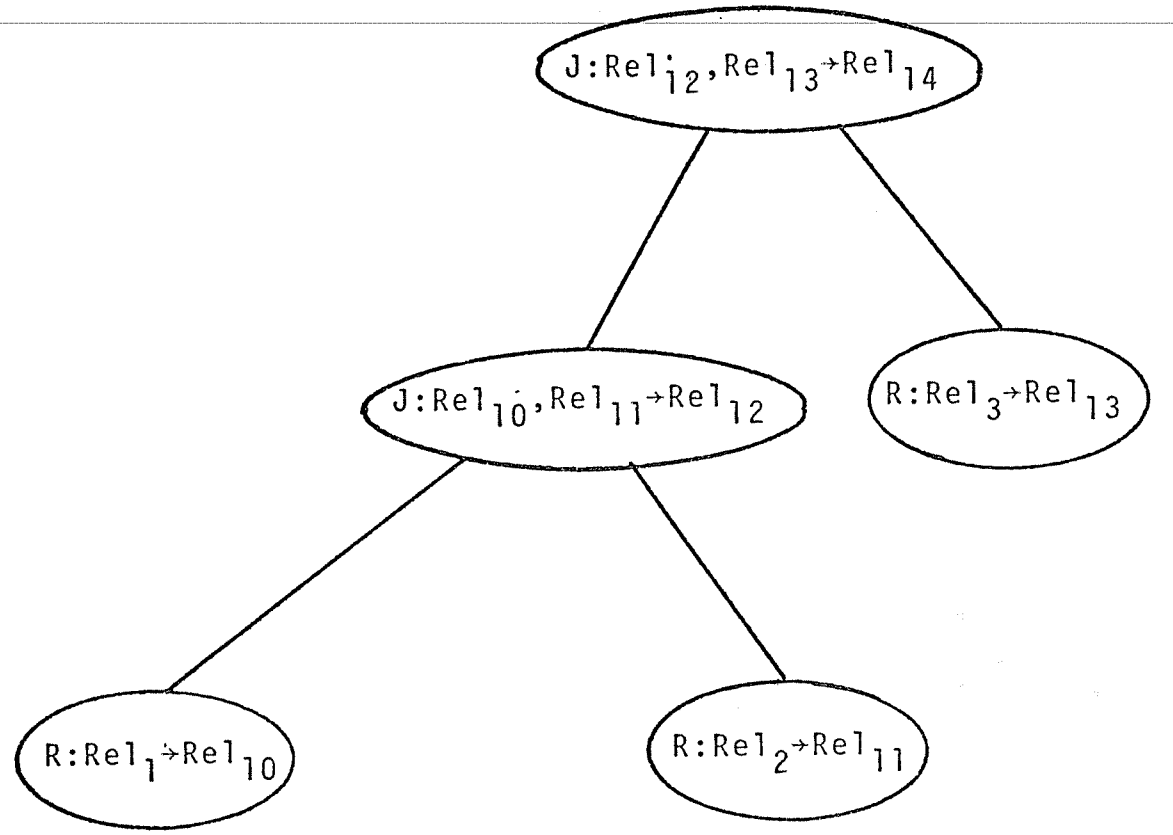


Figure 2.1
DIRECT Organization



R:Restrict
J:Join

Figure 2.2

A Sample Query Tree

2.2 Software

In this section we will describe those features of DIRECT which permit alternative strategies for the assignment of processors to instructions for execution. DIRECT is not a SIMD (Single Instruction stream, Multiple Data stream) machine such as RAP[4,6,7]. Different processors can simultaneously be executing different query packets, different instructions from the same packet, or different instructions from different packets. Furthermore, the number of processors assigned to an instruction can be increased or decreased during the execution of the instruction without modifying the instruction.

2.2.1 Join Algorithm

In a relational database system one of the most time consuming operations that must be performed is the join operator. In [8], 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 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 [8] 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.

The processing of joins using the "nested loops" algorithm and the ability to increase or decrease the number of processors assigned to an instruction requires that DIRECT be able to:

- supply a processor with all pages of the inner relation in sequence. This is the function of the GET_PAGE operator described below.
- keep track of which page of a relation (outer relation in the case of a join instruction) a free processor should be assigned to examine so that the processor does not examine a page which has been already given to another processor. To perform this function the NEXT_PAGE operator and the Query Packet Task Table are employed.

2.2.2 The NEXT_PAGE Operation

This is a request from processor to the back-end controller for the next available page of a relation. The use of the NEXT_PAGE operator is demonstrated in Figure 2.3 which implements the restrict operator.

```
procedure restrict(qpkt,relA,qpid:integer);
var       i: integer;
begin
     i:=NEXT_PAGE(qpkt,relA,qpid);
     while i <> -1 do
     begin {-1 indicates end of relation}
          readpage(i); {read page from CCD module i
                                into local memory}
          perform the restrict operation on the page;
          i:=NEXT_PAGE(qpkt,relA,qpid);
     end;
end;
```

Figure 2.3

It is very important to notice that this procedure can be executed by any number of processors simultaneously. Each processor, through the use of the NEXT_PAGE operator, will work on a distinct subset of the pages of relation relA. Furthermore, it is possible to add additional processors after the restrict is already partially executed. In addition, a processor can be removed before an operation is completed by having the back-end controller send it a fake "end-of-relation" signal in response to a NEXT_PAGE request. The processor will respond by first flushing its output buffer into a CCD memory module and then will request a new instruction from the back-end controller.

2.2.3 Query Packet Task Table

Since a query packet (or an instruction from a query packet) can be assigned to any number of processors, there must be a way to prevent two simultaneous NEXT_PAGE requests from different processors executing the same instruction from getting the same page. This is handled by the use of the Query Packet Task Table which is shown in Figure 2.4. The Query Packet Task Table has one entry for each instance of each relation referenced by each executing query. Associated with each entry is a monitor [9] which controls access to the table entry. Initially, the currency pointer for the relation is set to zero. Each NEXT_PAGE operation applied to an entry in this table is performed as an indivisible operation which returns the current value of the currency pointer and then increments the pointer by one. When the value of the currency pointer exceeds the size of the relation, all subsequent NEXT_PAGE operations on that entry receive an "end-of-relation" value. This insures that processors executing the same relational algebra operation will examine different pages of the source relation.

2.2.3 GET_PAGE Operation

The GET_PAGE operator is a request from a processor to the back-end controller for the address of the CCD memory module which contains a particular page of a relation. No monitor is needed to coordinate GET_PAGE requests. Figure 2.5 illustrates

QUERY PACKET TASK TABLE

QPKT#	RELNAME	CURRENCY POINTER	RELATION PAGE TABLE PTR
		• • •	

Figure 2.4

```
procedure join(qpkt,relA,relB,qpid: integer);
var i,j,k: integer;
begin
     i:=NEXT_PAGE(qpkt,relA,qpid);
     while i <> -1 do
     begin
          readpage(i);
          j:=1;
          k:=GET_PAGE(qpkt,relB,qpid,j);
          while k <> -1 do
          begin
               readpage(k); {read page of relB
                             from CCD module k}
               join page of relA with page of relB;
               j:=j+1;
               k:=GET_PAGE(qpkt,relB,qpid,j);
          end;
          i:=NEXT_PAGE(qpkt,relA,qpid);
     end;
end;
```

Figure 2.5

the use of this operator to perform a join of relations A and B.

2.2.4 Relation Fragmentation and the Compress Operator

As discussed in [3], a problem arises when more than one processor is used to select from a relation those tuples which satisfy a search condition. We call this problem "relation fragmentation". To illustrate we will use the restrict operator. Assume that the relation being restricted is divided into fixed size pages (tracks in RAP) and that there is one processor per page (track). Then, when the search condition is applied by all processors in parallel to the relation, each processor will produce a subset of the new relation. This new relation will contain those tuples which satisfy the restriction. In DIRECT each processor will produce some fraction of a page of the new relation. In RAP, each processor will "mark" those tuples on its track which satisfy the search condition. If this new relation represents the results of the query, this fragmentation is not a significant problem. However, if this new relation is to be used by a subsequent operator (such as a join) in the query, then the degree of relation fragmentation will have a significant impact on the performance of this operator. This performance degradation will occur because any subsequent operator which uses the fragmented relation as an operand will have to read all of the partially filled pages (tracks) in order to access all the tuples of the intermediate relation.

To solve this problem DIRECT employs a operator called

compression. The function of this operator is to read partially filled pages of a relation and output full ones. In [3], we analyze the use of this operator and develop a technique for estimating the optimal number of processors to perform the compression so that the benefits of its employment are maximized and the cost minimized.

3.0 Query Processor Assignment Strategies

In this section we will describe and analyze four alternative strategies for the assignment of processors to queries. These strategies appear to be viable for any MIMD database machine.

As we will demonstrate in Section 5.0, each strategy will provide a different level of system performance in terms of the execution time of a set of benchmarks which are described in Section 4.0.

3.1 SIMD Assignment

One of the original design objectives of DIRECT [1,2], was to avoid the SIMD nature of previous database machines such as RAP. In the SIMD assignment strategy, all processors are assigned to execute the same instruction from a single query simultaneously. When the current instruction has terminated, the back-end controller assigns the next instruction from the same query packet to all processors. This continues until the packet has terminated at which point the controller selects the next

query packet to execute.

The flaws of this strategy are fairly obvious. If, for example, the size of a source relation for some instruction is 10 pages (tracks) and there are 100 processors available, then 90 processors will be idle during the execution of the instruction.

3.2 Packet-Level Assignment

In this strategy, when the back-end controller decides to execute a query packet, the controller examines the query packet and attempts to estimate a priori the "optimal" number of processors to assign to the packet. Our estimation heuristic uses the number and size of the source relations referenced by the instructions in the packet, and the number and type of operators in the packet. However, once this value has been estimated it remains fixed throughout the execution of the query.

After the back-end controller estimates how many processors should be assigned to a packet, it examines the packet and selects an executable (enabled) instruction. An instruction is executable when its input relation(s) exist. Clearly, if the query is in a tree format all leaf nodes are immediately executable. A node higher up in the tree is enabled whenever all of its descendents have finished executing.

Let QPS represent an estimate of the "optimal" number of processors to be assigned to the query packet. If the instruction selected for execution is a restrict operation, then the controller will assign MIN($|S_i|$,QPS) processors to the instruction where $|S_i|$ is the number of pages in S_i, the source relation to be restricted. If the operation is a join of relations S_i and S_j, then MIN(MAX($|S_i|$,$|S_j|$),QPS) processors are assigned. Selecting the larger of the two relations S_i and S_j means that, if MAX($|S_i|$,$|S_j|$) $\leq$ QPS, each processor will join one page of the larger relation with every page of the smaller relation. This appproach maximizes the degree of concurrency and hence minimizes execution time.

At this point, if all the processors assigned to the packet have not been utilized, the next executable instruction from the packet is initiated. This continues until either all the processors assigned to the packet are executing some instruction from the packet, or until no more executable instructions are available (since their inputs have not been generated). If no more executable instructions exist, the available processors are placed on an idlelist associated with the packet until an instruction is enabled. Idle processors are not reassigned to another packet before the packet terminates.

If, when the first instruction in a packet is initiated, the number of processors available is less than QPS, the packet is still initiated. Then, when another executing packet terminates, the additional processors are assigned to the sub-optimal packet. At this point the processors will either be assigned to an instruction which is below its optimal level or, if no sub-optimal instructions exist, to another instruction in the packet which is executable but has not been yet initiated.

3.3 Instruction-Level Assignment

For this strategy, scheduling and processor assignment is performed on an instruction by instruction basis. The optimal number of processors assigned to an individual restrict or join instruction is limited only by the number of processors available. If the total number of processors available is MAXQPS, then for a restrict operations QPS=MIN(|S_i|,MAXQPS) and for a join QPS=MIN(MAX(|S_i|,|S_j|),MAXQPS).

When a processor becomes idle, the back-end controller first attempts to assign the processor to any executing instruction which does not have its optimal number of processors. If no sub-optimal instructions exist, the processor is assigned to an enabled instruction from a query packet which is currently being executed. If there are no enabled instructions from the currently executing packets, then a new packet is initiated. If there are no packets awaiting execution, then the processor is placed on an idle list until a new packet arrives from the host or an instruction is enabled.

3.4 Data-flow Assignment

This strategy is a variation of the instruction-level assignment strategy except for the criterion which is used to enable instructions in the query packet tree. In the sections below, we will first explain the basic concepts of data-flow machines. Then we will describe processor assignment and query

execution in the data-flow assignment strategy.

3.4.1 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 memory section and the processing section. 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 execution of an instruction is sent via the interconnection device to one or more memory cells, possibly enabling one or more instruction(s) in the destination cell(s).

Data-flow architecture seems particularly attractive when we consider the fact that many programs contain a significant degree of inherent parallelism that cannot always be exploited by conventional machines or even multiprocessors. Furthermore, in order to exploit the parallelism, extensive analysis of the program must be performed by a compiler. In a data-flow machine this is unnecessary since the notion of sequentiality in the execution of programs does not exist.

Various architectures for data-flow machines have been proposed [10-13]. 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 [12] talks about assigning such instructions as add and multiply to the processors whereas Arvind [10] and Rumbaugh [13] 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 on which processors operate can be the a whole relation, a fragment of a relation, or a single tuple.

Thus it follows that the strategy described in Section 3.3 is basically data-flow. However, because of the memory unit size and the processor capabilities in DIRECT, any advantage claimed for a data-flow strategy will not show. A strategy using a fragment of a relation, in this case a physical page, is the fourth strategy we investigated. Its description is presented below.

3.4.2 Data-flow Assignment using Page-level Granularity

In this strategy a page of a relation is the basic unit on which processors operate and which is used for scheduling decisions. This means that an instruction can be initiated as soon as at least one page of each participating relation exists. Assigning processors to operate on pages rather than relations offers the possibility of having a very flexible processor allocation strategy. Furthermore, it becomes possible to distribute processors across all nodes of the query tree and to pipeline pages of intermediate relations between them. This will reduce

page traffic between the CCD memory and the mass storage device(s) because after a page of an intermediate relation is produced by one processor it will be read by the processor executing the subsequent operator.

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 suggested by Smith and Chang[14] and Yao [15]. There are, however, several important differences between the two strategies. The first deals with the number of processors which can be used to execute each node (relational algebra operator) in the query tree. In the pipelined strategy, 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. With the data-flow strategy 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 strategy we never need to wait for one node to completely finish before initiating the subsequent operator as has been suggested is necessary for pipelining [15].

Since in the data-flow strategy we initiate join operations before the operand relations are completely computed, we cannot accurately predict what their final sizes will be. Consequently it is difficult to intelligently choose the outer relation (see Section 2.2.1). Our solution is to begin both restrict opera-

tions with the same initial processor allocation. Then, after two pages of each relation have been produced, we estimate the final size of each relation by considering the densities of the output pages, the makeup of the source relations, and the fraction of the source relations that went into producing these pages. We can now pick the outer relation and alter the scheduling policy for each restrict.

We clearly want to allocate a large number of processors to the restrict producing the outer relation. This will cause a large number of output pages to be produced. Therefore a high level of concurrency can be attained in the execution of the subsequent join by assigning a processor to each page of the outer relation. Each will then join its page with the entire inner relation.

On the other hand, since each page of the outer relation will be joined with the entire inner relation, the inner relation will be resident in the CCD memory for the duration of the join and consequently should be as small as possible. In the SIMD, Packet-based, and Instruction-level strategies a large number of processors can be allocated to the restrict producing the inner relation in order to minimize its execution time. Then, after both relations have been completely computed, the inner relation is selected and compressed to a manageable size using the compress operator.

Instead of using the compress operator with the data-flow strategy we employ the following technique for generating the

inner relation. Since we want pages of the inner relation to be consumed by the processors executing the join fairly soon after they are produced (so that they are not paged out of CCD memory) and since joining two pages takes considerably longer than restricting one page, we want pages of the inner relation to be produced at a slower rate than pages of the outer relation. Therefore, by allocating a small number of processors to this restrict, we can achieve the goal of producing a small number of relatively full pages without employing the compress operator.

The number of processors assigned to a restrict operator is a fraction of the number of pages of the relation which is to be examined. For the first phase (before estimating the final size) the fraction was chosen to be one half (thus the restrict of a relation with 20 pages would be assigned 10 processors). After the outer and inner relations are selected, the number of processors assigned to the restrict producing the inner relation is reduced from one half to one tenth so that the inner relation will not be significantly fragmented. No change is made to the number of processors assigned to produce the outer relation. Joins are allocated as many processors as there are pages in the outer relation.

4.0 Design of an Experiment to Compare the Four Strategies

In order to evaluate the four processor assignment strategies and to uncover any unforeseen problems with the design of DIRECT, a detailed simulation of DIRECT was implemented. (Actually there are four variations: one for each strategy. However, since the differences are minor we will subsequently refer to them as one.) This simulation permits us to make detailed performance measurements on all aspects of the system including both resource utilization (processors, CCD memory modules, and secondary memory) and software performance (query execution and memory management).

In Section 4.1, the hardware and software characteristics we assumed will be described. Then, in Section 4.2, the set of tests which were used to evaluate the different processor assignment strategies will be outlined.

4.1 Hardware and Software Characteristics

4.1.1 Processors and Query Execution

The performance characteristics of a processor are based on the instruction execution times of a PDP LSI-11/03. There are three main operations which depend on these execution times:

- The time required to transfer a page between a CCD memory module and the local memory of a processor.
- The time required to determine if a tuple from a relation satisfies a selection criterion.
- The time required to join two tuples.

The time to transfer a page between a processor's main memory and a CCD memory module was assumed to be 33ms based on an LSI-11 Q bus bandwidth of $0.5 * 10^6$ bytes/second [16] and a page size of 16K bytes.

The following restrict algorithm is employed by a processor on a page of a relation. First a pointer is set to the appropriate attribute of the next tuple. Since all tuples in a given DIRECT relation have the same fixed length, this operation takes a constant amount of time. Next the attribute is compared against the restrict criterion. We assume that each attribute is a character string and that if the tuple does not satisfy the restrict criterion three tenths of the characters in the attribute are examined before a match failure occurs. Otherwise, the full attribute is examined. For each restrict operator the fraction of tuples which satisfy the restrict condition was chosen from an exponential distribution with minimum of 0, maximum of 1, and mean of 0.125 (chosen to produce result relations with a reasonable size). When a tuple satisfies the search condition, the desired attributes are moved character by character to an output buffer in the processor's main memory.

The join algorithm used by each processor is the nested loops algorithm [8]. We assume that, on the average, three tenths of the characters of the joining attributes from both tu-

ples are compared before a failure is determined. If the join is successful, the desired attributes are selected and moved to an output buffer. For each join the fraction of tuples (of the product of the number of tuples in both pages) which satisfy the join qualification was selected based on exponential distribution with minimum of 0, maximum of 1, and a mean of 0.0035 (again chosen to produce reasonably sized result relations).

4.1.2 CCD Memory Modules and Interconnection Matrix

The bandwidth of an individual memory module was assumed to be 2 megabytes/second based on INTEL 2314 CCD chips. This implies that a 16K byte page could be transferred into (from) a CCD memory module in 8.2 ms if the transfer rate was not limited by the LSI-11 Q bus or the disk transfer rate (see Section 4.1.3). Furthermore, we assume that the interconnection matrix does not impact CCD memory performance (see [1],[2]).

4.1.3 Mass Storage Devices

Since DIRECT is a virtual memory machine, pages of relations which are not being referenced by an active query packet, are resident on one or more mass storage devices. When a page is subsequently referenced it is loaded into a CCD memory module. IBM 3330 disks were used as the model for our mass storage devices The transfer time for a 16K byte page is 20ms. The time to seek one track is 0.148ms and the latency time is 8.4 ms. We assume that there are two disks available for relation storage and swapping.

The simulation services disk requests in a first-come first-served order. Therefore, when a processor issues a request for a page of a relation, the length of time required to satisfy the request is dependent on the other disk activity. The directory structure used in the simulation models each relation as occupying a set of adjacent tracks on one disk. The pages of a relation are first spread across all tracks in one cylinder before the next cylinder is used in order to minimize seek times.

4.2 Experiment Design

The database used to evaluate the four processor assignment strategies consists of 15 relations. The size (in pages) of each relation was randomly chosen from an exponential distribution with a mean size of 23 pages, minimum relation size of 1 page, and a maximum relation size of 100 pages. The tuple length of each relation was chosen from an exponential distribution with a mean size of 55 bytes, a minimum size of 10 bytes, and a maximum size of 100 bytes. The total database size is 5.5 megabytes. Appendix I contains detailed information on each relation.

Six different sets of queries were chosen to evaluate the alternative processor allocation strategies. We will describe below how each query was generated. Classes I to IV each contain five query packets and correspond to a range of overhead intensive queries (Class I) to execution intensive queries (Class IV) [17]. Tests Mix I and Mix II each contain ten query packets and

represent what we feel are a reasonable mix of the different classes of queries. Table 4.1 summarizes these six different experiments.

TABLE 4.1

Testname	Number of Queries	Description
Class I	5	Each with 1 restrict only
Class II	5	Each with 1 join & 2 restricts
Class III	5	2 Queries: 2 joins & 3 restricts 3 Queries: 3 joins & 4 restricts
Class IV	5	2 Queries: 4 joins & 5 restricts 1 Query: 5 joins & 6 restricts 1 Query: 6 joins & 7 restricts 1 Query: 7 joins & 8 restricts
Mix I	10	2 Queries from Class I 3 Queries from Class II 3 Queries from Class III 2 Queries from Class IV
Mix II	10	1 Query from Class I 4 Queries from Class II 4 Queries from Class III 1 Query from Class IV

We feel that Classes II and III contain the types of queries which are typically performed by users in accessing relational databases. This is why Mix I and Mix II include a high percentage of queries from these two classes. If views are supported by the relational database system and queries are modified according to the view, it has been observed [18] that it is not unusual for a modified query to contain five to seven join operations. Therefore, we feel that the results of Class IV may be as

significant as the results of Mix I and Mix II because Class IV is a very execution intensive test with a large number of joins in each query packet.

5.0 Experiment Results

5.1 Establishment of a CCD Memory Module to Processor Ratio

The first test performed served two functions. Its primary function was to establish an appropriate ratio of CCD memory modules to query processors. Once this value was established it would be used in all subsequent tests. The second function of this experiment was to determine how the performance of each processor assignment strategy is affected by this ratio. We felt that this should provide some indication about how efficiently each strategy uses the CCD memory modules available.

To perform this experiment we fixed the number of processors available at 50 and then ran test Mix I on all strategies for five different CCD memory sizes: 50, 100, 150, 200, and 250. Figure 5.1 contains the results of this experiment. As the reader will notice the performance of the SIMD, packet-level, and instruction-level strategies continue to improve as the number of CCD memory modules increases. If the number of CCD memory modules is increased beyond 250, this trend continues until enough modules are present so that pages from source, intermediate, and final relations never have to be ejected to secondary memory. The data-flow strategy, on the other hand, is not significantly

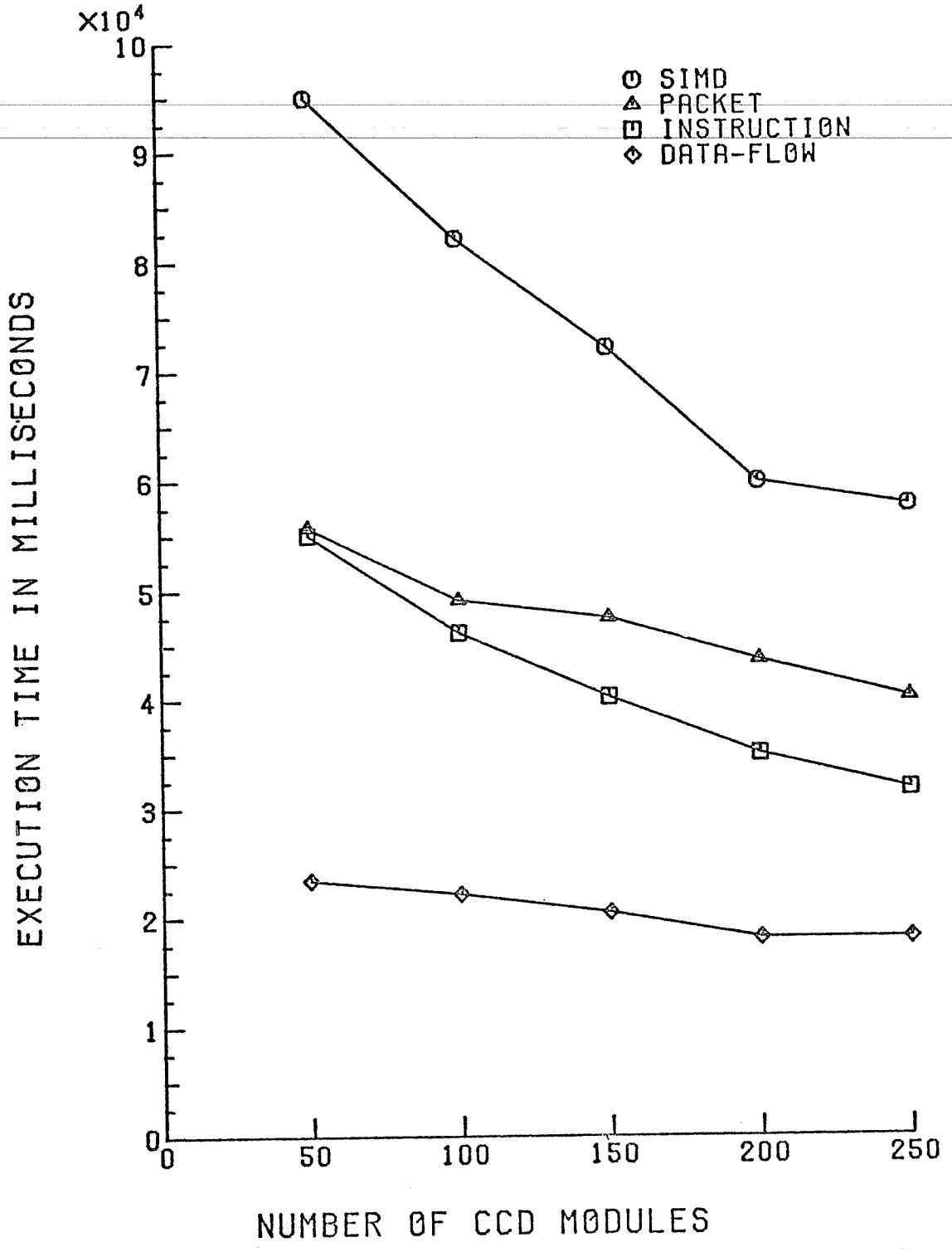


Figure 5.1

Effect of CCD to PROCESSOR Ratio for 50 Processors

affected by the number of CCD memory modules present. This result seems to indicate that this strategy indeed succeeds at using pages from intermediate relations before they are paged out. This saves a write operation to mass storage followed by a read operation when an intermediate relation is subsequently accessed.

We feel that the reason that the packet-level strategy is less affected by an increase in the number of modules available than the instruction-level strategy is that it is less flexible and hence has better locality properties than the instruction level strategy. While this argument should also apply to the SIMD strategy the results indicate otherwise. Therefore, there may be another, yet undiscovered, reason why the packet-level strategy behaves this way.

Although not presented, similar results were obtained at several other levels of processors and for other tests. Thus, as a compromise between the thriftiness of the data-flow strategy and the greediness of the SIMD, packet-level, and instruction-level strategies, we choose a CCD memory module to processor ratio of 2:1. This ratio was used in all subsequent tests. Therefore, in the results presented below, if there are n processors available there are 2*n CCD memory modules available.

5.2 Analysis of the Simulation Results

Using this 2:1 ratio, each of the six tests (Classes I through IV and Mixes I and II) was executed using each alternative strategy for a range of available processors from 1Ø to 1ØØ in steps of 1Ø. The results of these experiments are presented in Figures 5.2 through 5.7. We will interpret these graphs below. However, the schedule of packets and instructions for any given processor level is not necessarily an optimal schedule. At any given point in time, when the back-end controller makes a decision regarding which instruction a processor should be assigned to or which page should be ejected from CCD memory, it chooses the "best" option. This local (immediate) optimization does not always produce an optimal schedule. Consequently, certain anomalies can occur in the results. For example, in Class I (Figure 5.2), as the number of processors available increases from 1Ø to 2Ø, the execution time of the SIMD strategy increases instead of decreasing as expected. There is certainly some schedule of packets for this case in which the execution time either decreases or remains constant as the number of processors is increased from 1Ø to 2Ø. It is simply the case that the back-end controller made a decision that turned out in the long run to be a bad decision.

One obvious result from these experiments is that the SIMD strategy always performs significantly poorer than all the other strategies. Our interpretation of this is that any database

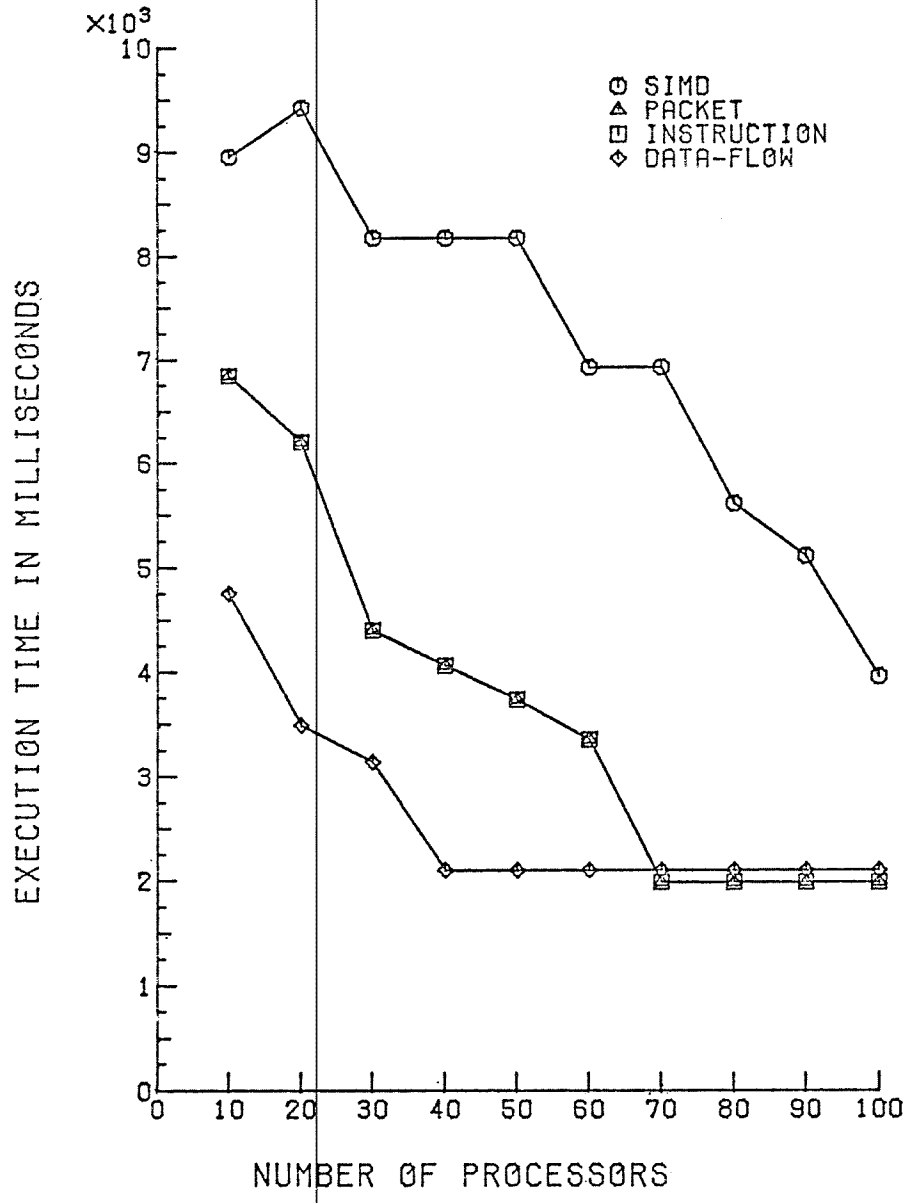


Figure 5.2

Class I

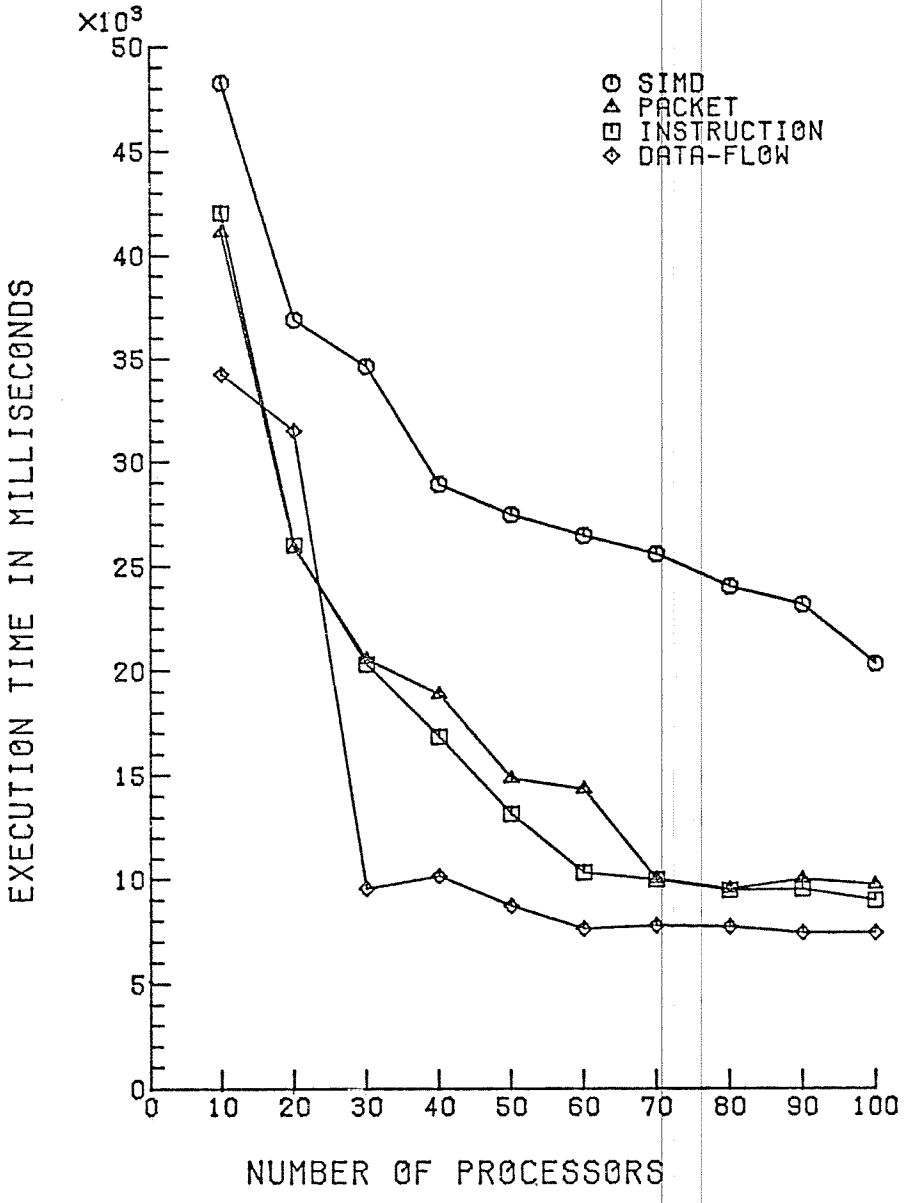


Figure 5.3

Class II

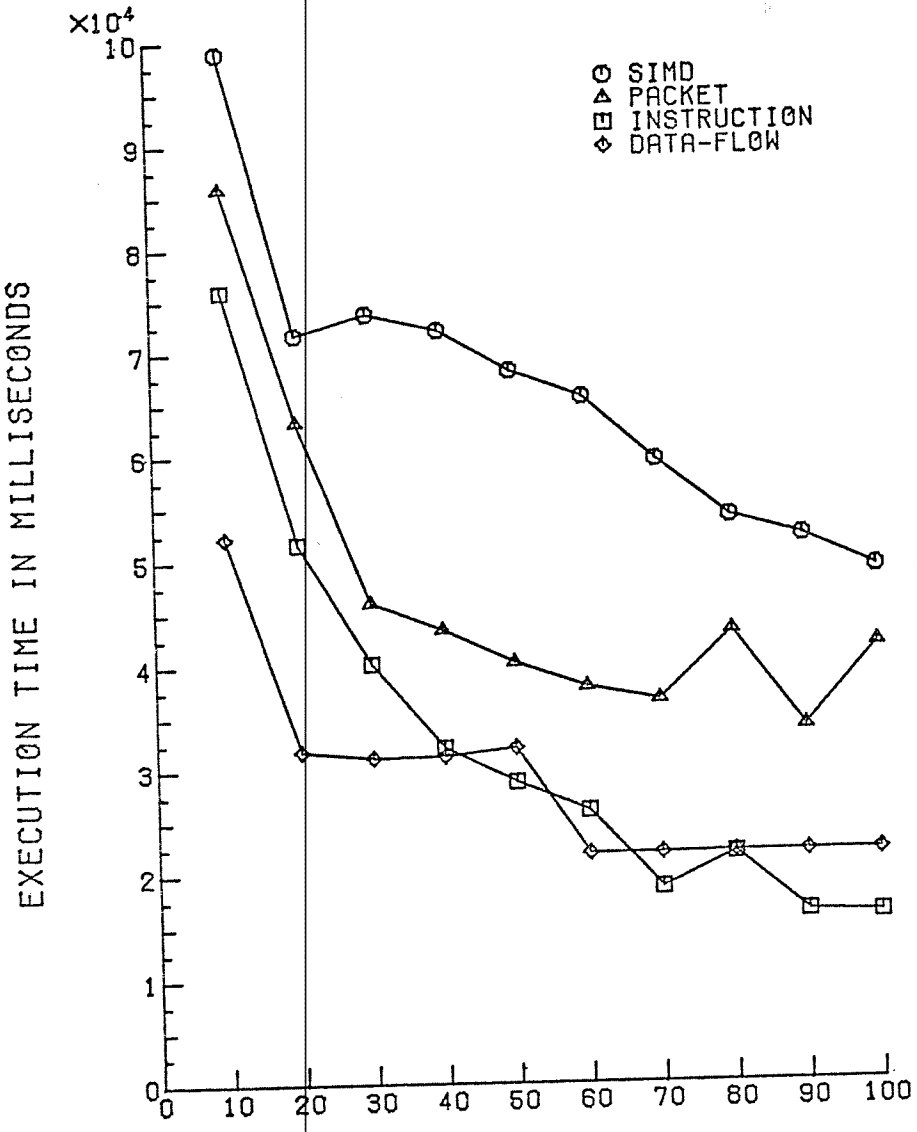


Figure 5.4

Class III

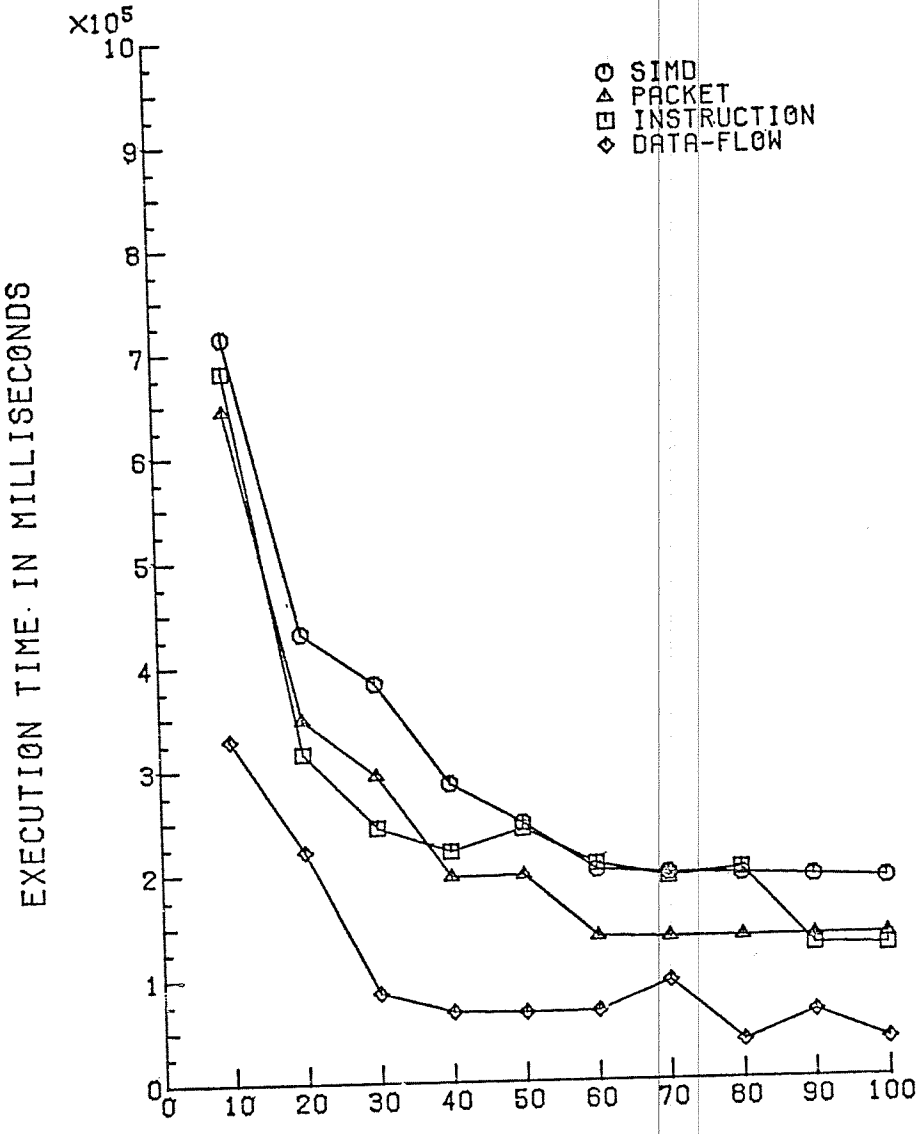


Figure 5.5

Class IV

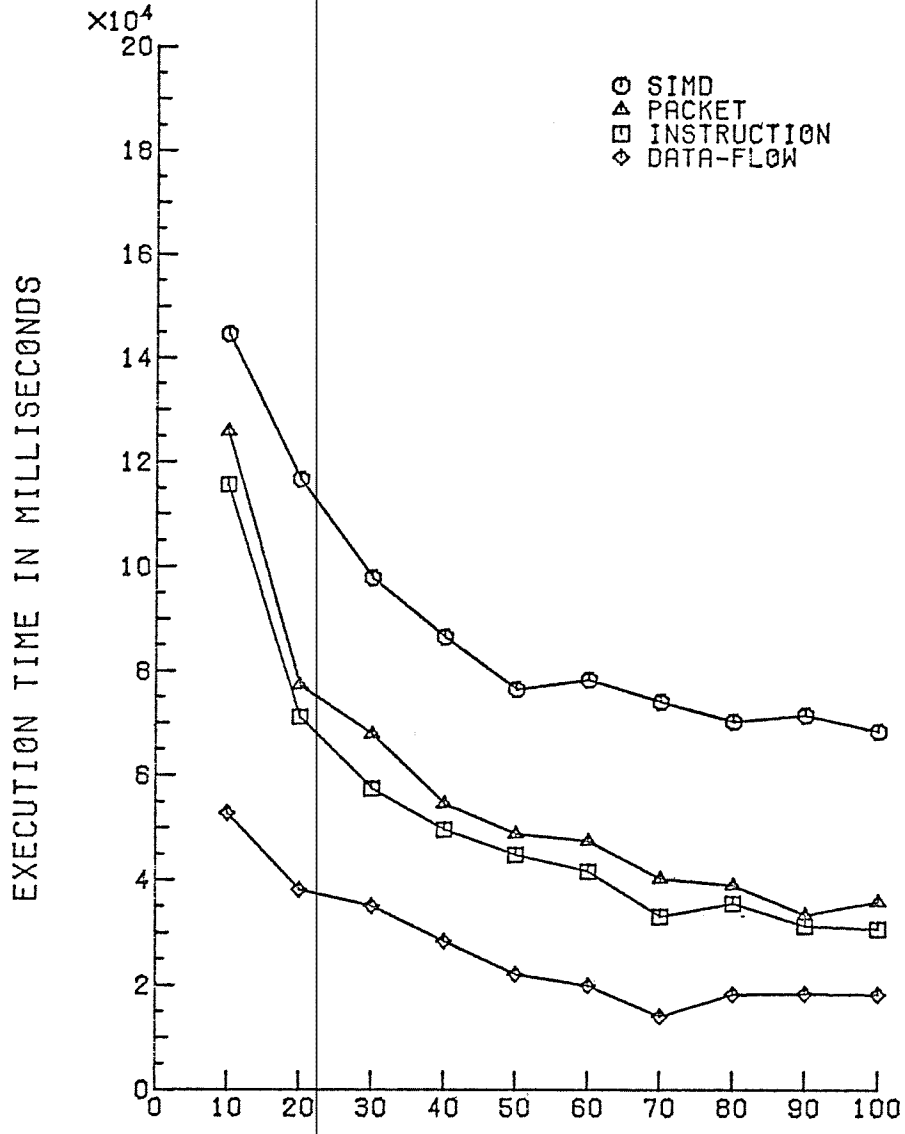


Figure 5.6

Mix I

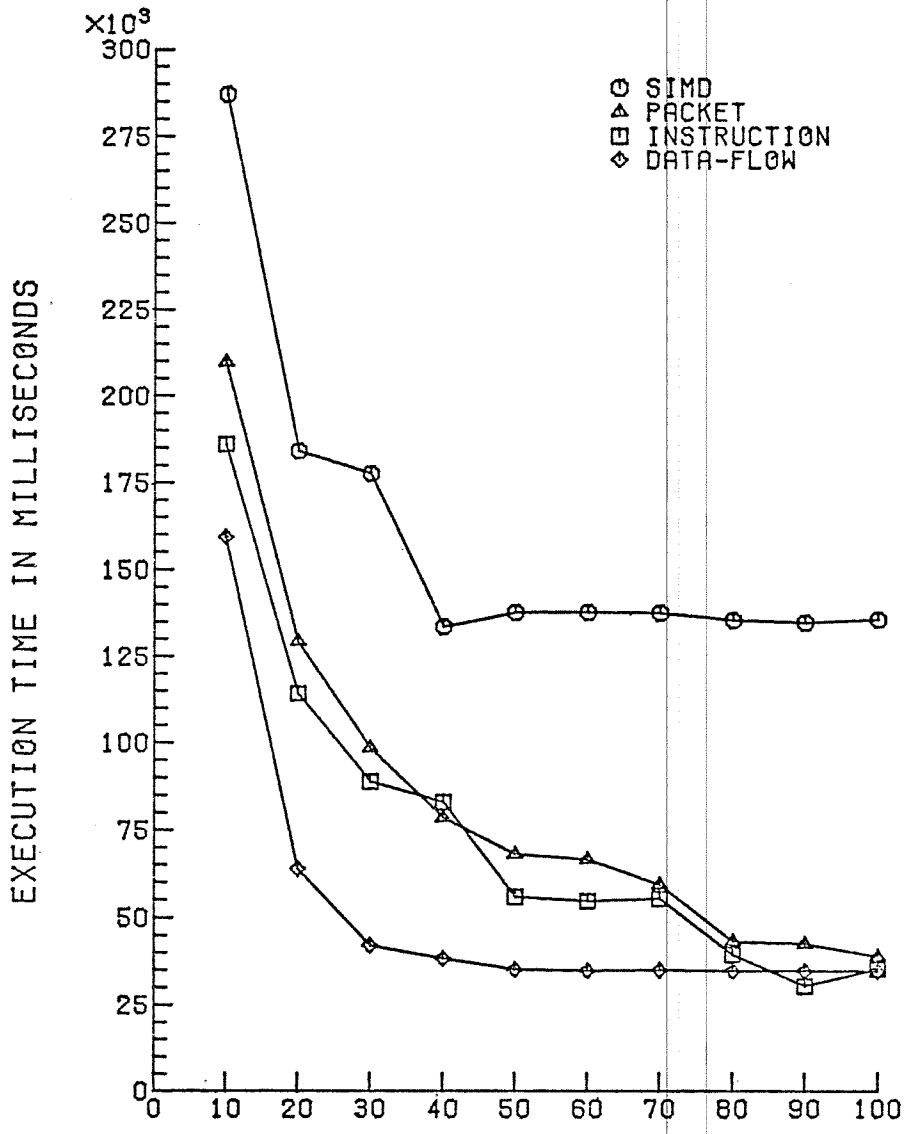


Figure 5.7

Mix II

machine which does not employ one of the MIMD strategies is most likely a poor design.

One surprising result illustrated by these tests is the relatively good performance of the packet-level strategy when compared with the instruction-level strategy. The execution time of Class I for both is identical because each query packet contains only one instruction. For Class II their performance is very similar. Class III is the only case where the instruction-level strategy is significantly better than the packet-level strategy. On Class IV, over the range of 40 to 90 processors, the packet-level strategy actually outperforms the instruction-level strategy. This apparently occurs because the packet-level strategy thrashes less. In the packet-level strategy, as a processor finishes executing an instruction it is either reassigned to another instruction in the same packet or is left idle until an instruction in the packet is enabled. Under identical conditions the instruction-level strategy would assign the free processor to an instruction from another packet or even initiate a new packet. Executing this new instruction will probably result in pages from secondary memory replacing pages currently in the CCD memory. As a consequence when an instruction from the original packet is finally enabled, its operands will most likely have been paged out. Finally, for Mixes I and II the performance of the instruction-level strategy is about 10% better than that of the packet-level strategy.

These tests clearly indicate the superiority of the data-

flow strategy for processor allocation. In all tests ranging from overhead-intensive (Class I) to execution-intensive (Class IV) and the two mixes (Mix I and II) the data-flow strategy always performed significantly better than any of the other three strategies. If Mix I and Mix II are taken to be representative of typical query mixes, then, for a given number of processors and CCD memory modules, the data-flow strategy was approximately four times as fast as the SIMD strategy and about twice as fast as the instruction-level and packet-level strategies. Furthermore, when one examines the performance of each strategy under heavy loads (less than 50 processors available), the data-flow strategy demonstrates an even greater performance improvement.

5.3 Effect of Swapping on Performance

The next experiment that we conducted was to determine the impact that swapping pages between CCD memory modules and secondary memory has on query execution time. We felt that this was a very significant experiment because it has been argued that database machines which use paging will always be I/O bound [19].

To determine the effect of swapping on system performance we modified the simulations for the data-flow and instruction-level strategies so that the time to transfer a page between a CCD memory module and a disk is 0 ms. In this way it appears that the bandwidth of the channel and disk are infinite. The size of the CCD memory is still limited, however, by the 2:1 ratio. Then we reran test Mix I for the data-flow and instruction level stra-

tegies. Figures 5.8 and 5.9 present the results of this experiment.

For the instruction-level strategy (Figure 5.8), the improvement averaged over all processor levels is 33%. Thus swapping has the effect of decreasing system throughput about one third. While significant, we feel that the overhead of swapping is not as high as expected.

It is more difficult to measure this value for the data-flow strategy since the results presented (Figure 5.9) include another scheduling anomaly. That is, in the processor range of 50-80 processors the standard data-flow strategy actually performed better than the data-flow version with infinite I/O bandwidth. Ignoring this range of processors, the improvement achieved with infinite I/O bandwidth is only 14%. This figure certainly reinforces the apparent superiority of the data-flow strategy and should indicate that virtual memory database machines can be organized in such a way to avoid being I/O bound as has been claimed.

5.4 Message Activity

While we have not yet modeled the back-end controller requirements for each strategy, we have modeled the message activity of each strategy. Each operator which is sent to a processor is counted as one message. In the SIMD strategy, the distribution of the operator to all processors is counted as only one message since we assume that such a database machine would have a broadcast capability. Each relation page request executed by a processor is counted as three messages: one to the back-end controller to make the request, one to the processor from the controller containing the CCD memory module number, and another from the processor to signal that it has read (written) the memory module so that the controller can free the page frame. When "end-of-relation" is received on a NEXT_PAGE or GET_PAGE request (i.e. there are no more pages available), only two messages are exchanged.

In Figure 5.10 we have plotted the number of messages sent between the set of processors and the back-end controller for all strategies running test Mix I. These results again illustrate the superiority of the data-flow assignment strategy. Although we were initially puzzled by the results (in fact we expected the data-flow strategy to have the highest message traffic) a little reflection provided an explanation. The reason why the message traffic for the three non-data-flow strategies increases as the number of processors increase is due to the problem of relation fragmentation. When the number of processors employed is 10, all four strategies have about the same amount of message traffic. This is expected because the degree of relational fragmentation is low and hence the cost of compression is also low. However, as the number of processors available to execute each operator increases, each intermediate relation tends to be more and more fragmented and hence the compress operator needs to read more and more fragmented pages. This increases the number of NEXT_PAGE

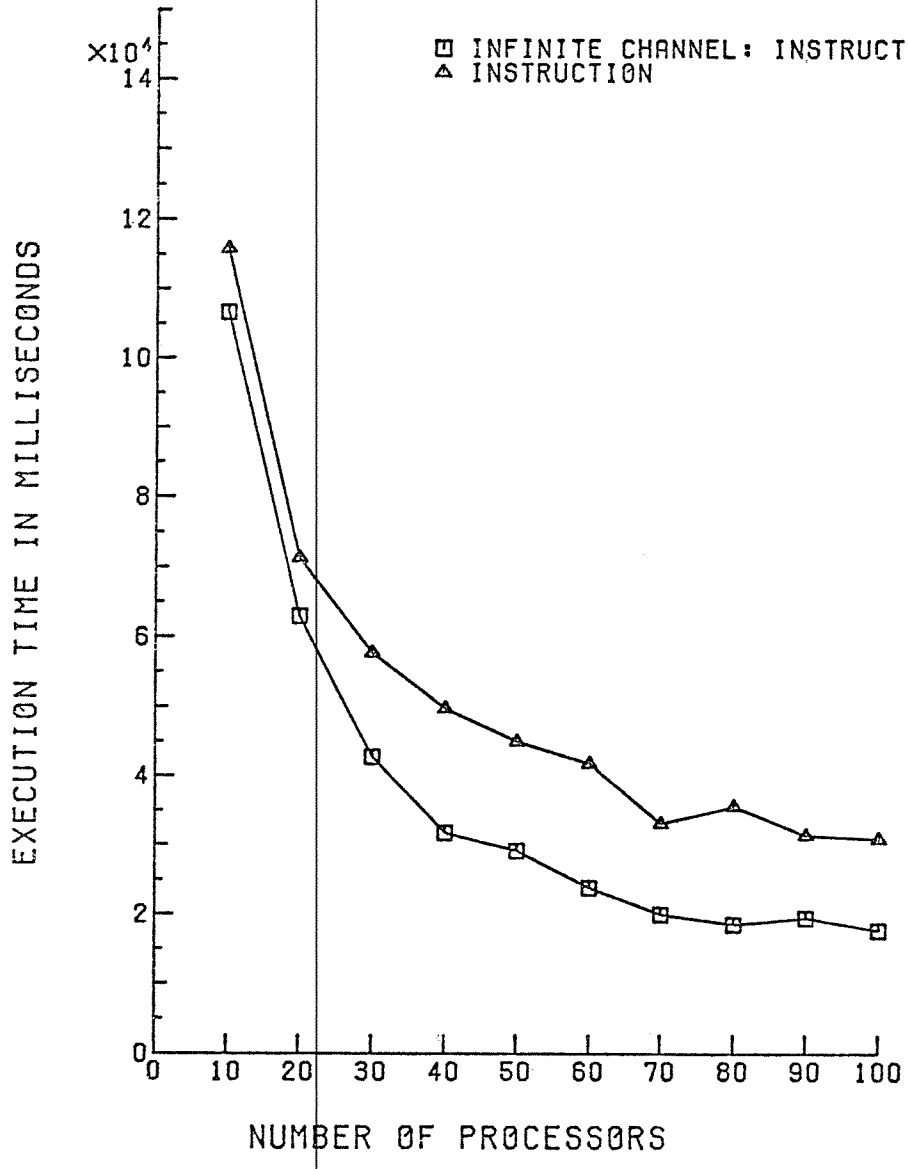


Figure 5.8

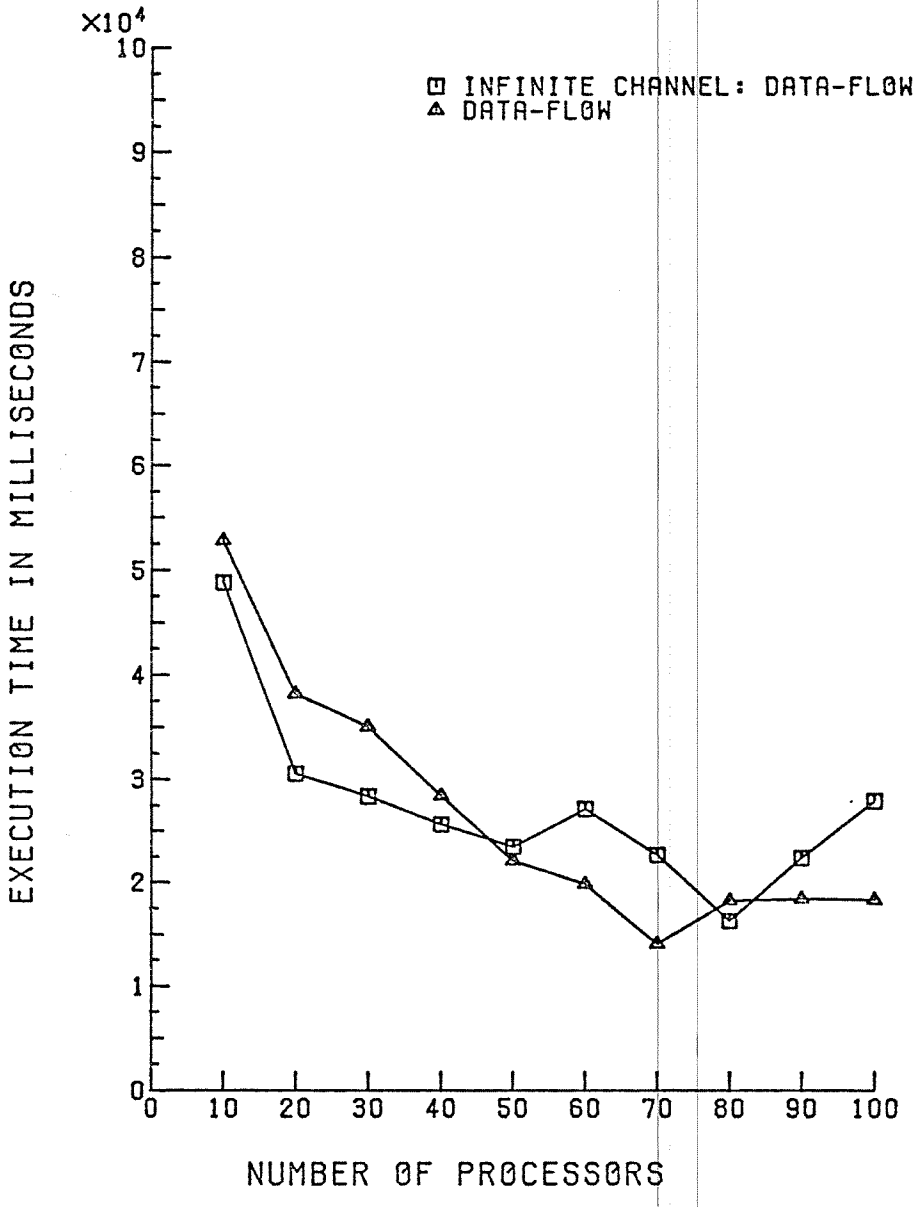


Figure 5.9

Effect of Infinite Capacity Channel and Disk for Mix I

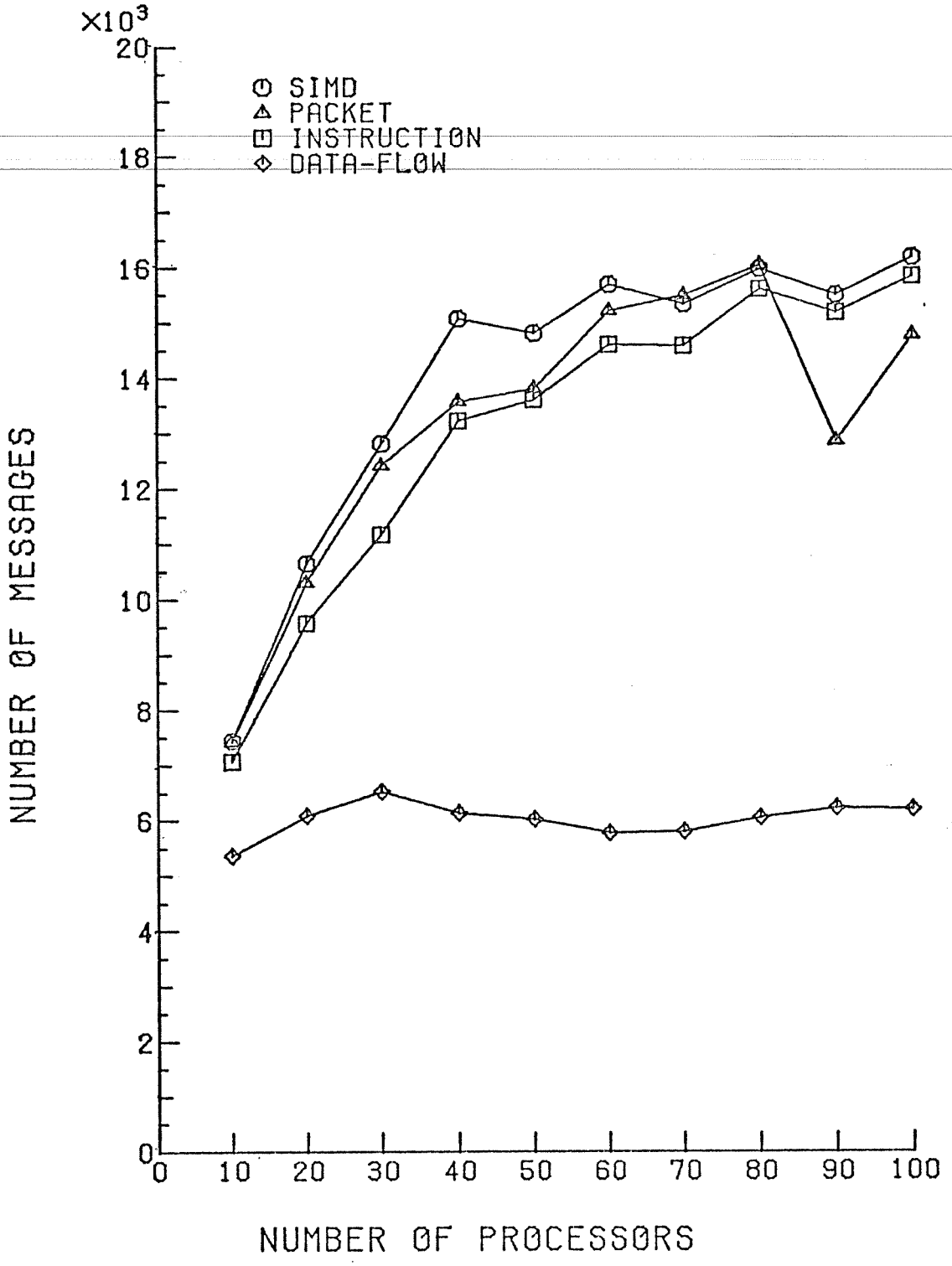


Figure 5.10

Message Activity for Mix I

operations performed to execute the packet and hence the overall message traffic also increases. Furthermore, since with the data-flow strategy there is no need to compress pages, there is no message traffic generated in distributing the compress operator to the query processors for execution.

6.0 Conclusions and Future Research

In this paper we have described and evaluated four alternative strategies for assigning processors to queries in multiprocessor database machines. As originally predicted in [1], our results demonstrate that SIMD database machines are indeed a poor design when their performance is compared with that of all the MIMD strategies we have presented.

We have also introduced the application of data-flow machine techniques to the processing of relational algebra queries. As illustrated by every experiment conducted, the data-flow strategy is clearly superior to the other strategies we have described. Furthermore, our results indicate that a two-level storage hierarchy (in which relations are paged between a shared data cache and mass storage) does not have a significant impact on performance if the data-flow query processing strategy is employed. Since the data-flow strategy is a generalization of the pipelined strategy, our results verify the expected performance of the pipelined strategy as was anticipated by Smith and Chang [14] and Yao [15].

One significant problem exposed by this research is the high

level of message traffic activity. Even when the best processor assignment strategy is employed the amount of message traffic which must be supported is relatively high. If 8000 back-end controller instructions are required to process each of the 6000 messages passed in executing Mix I, (a figure derived from UNIX pipe code efficiency [20]), then 48 million instructions will be executed just to process the messages. If each instruction takes one microsecond then 48 seconds will be required to process the messages while only 20 seconds are required to execute the query.

To solve this problem we are following two strategies. For our initial implementation of DIRECT we are considering implementing message handling software in microcode on the back-end controller. This should help significantly. In addition we have just completed a preliminary design of a new data-flow machine architecture for processing relational algebra queries [21]. We intend to further investigate and evaluate such machine organizations so that we can efficiently support the data-flow assignment strategy.

7.0 Acknowledgements

Kevin Wilkinson very willingly listened and commented on ideas at various stages of the research that led to this paper. Both he and Dina Friedland have made many helpful suggestions and pointed out some problems with an earlier draft. For this, and their company in working on DIRECT, we wish to thank them.

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. Ozkarahan, E.A., Schuster, S.A., and Sevcik, "Performance of a Relational Associative Processor," ACM Transactions on Data Base Systems, Vol. 2, No. 2, June 1977, pp 175-195.

5. Stonebraker, M.R., Wong, E., and P. Kreps, "The design and implementation of INGRES," ACM Transactions on Data Base Systems, Vol. 1, No. 3, Sept 1976, pp.189-222.

6. Ozkarahan, E.A., Schuster, S.A., and K.C. Smith, "RAP - An associative processor for database management," Proceedings of the 1975 NCC, pp.379-386.

7. Schuster, S.A., Ozkarahan, E.A., and K.C. Smith, "A virtual memory system for a relational associative processor," Proceedings of the 1976 NCC, pp. 855-862.

8. 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.

9. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM Vol. 17, No. 10, Oct. 1974, pp. 549-557.

10. 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.

11. 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.

12. 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.

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

14. 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.

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

16. Digital Equipment Corportation, "Microcomputer Handbook," 1977.

17. Hawthorn, P., and M. Stonebraker, "Performance Analysis of a Relational Data Base Management System," Proceedings of the ACM-SIGMOD 1979 International Conference on Management of Data, May 1979, pp. 1-12.

18. Lorie, R.A. - Personal Communication.

19. Stonebraker, M. - Personal Communication.

20. Chesson, Greg - Personal Communication.

21. Boral, H. and D.J. DeWitt, "Design Considerations for Data-flow Database Machines," submitted to The 7th International Symposium on Computer Architecture, also Computer Sciences Technical Report No. 369, University of Wisconsin, September 1979.

APPENDIX I

TEST DATABASE

RELATION	SIZE IN PAGES	TUPLE WIDTH IN BYTES
1	27	41
2	36	41
3	40	57
4	34	20
5	28	99
6	5	63
7	12	39
8	7	11
10	8	41
11	20	38
12	38	37
13	40	42
14	10	14
15	9	77