Implementation of the Database Machine DIRECT

Haran Boral
David J. DeWitt
Dina Friedland
Nancy F. Jarrell
W. Kevin Wilkinson

Computer Sciences Technical Report #442

August 1981

Implementation of the Database Machine DIRECT

Haran Boral
David J. DeWitt
Dina Friedland
Nancy F. Jarrell
W. Kevin Wilkinson

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin

This research was partially supported by the National Science Foundation under grant MCS78-01721, the United States Army under contracts #DAAG29-79-C-0165 and #DAAG29-80-C-0041, and the Department of Energy under contract #DE-AC02-81ER10920.

ABSTRACT

DIRECT is a multiprocessor database machine designed and implemented at the University of Wisconsin. This paper describes our experiences with the implementation of DIRECT. We start with a brief overview of the original machine proposal and how it differs from what was actually implemented. We then describe the structure of the DIRECT software. This includes software on host computers that interfaces with the database machine; software on the back-end controller of DIRECT; and software executed by the query processors. In addition to describing the structure of the software we will attempt to motivate and justify its design and implementation. We also discuss a number of implementation issues (e.g., debugging of the code across several machines). We conclude the paper with a list of the "lessons" we have learned from this experience.

1. Introduction

DIRECT is a relational database machine whose design was begun in 1977. A detailed description of the proposed hardware organization of DIRECT is presented in [DeWi79a]. Related research results can be found in [DeWi79b,Bora81a,Bora80a]. The purpose of this paper is to describe the implementation of DIRECT and our experiences during the implementation. We will discuss the present hardware organization, the design of the software, and the lessons that we have learned by working on this project. This should not be considered a post-mortem, however. Work continues and DIRECT is still evolving.

Numerous database machine designs have been proposed since the early 1970's and a number of distinct approaches to the design of such machines have been identified (see [Hawt81a] and [Hawt81b] for two different classifications). However, despite approximately a decade of active interest, very few systems have been implemented.[1] Presently, (to our knowledge) there are three operational database machines: the IDM 500 from Britton-Lee Inc. [Epst80a], CAFS from ICL Ltd. [Babb79a], and DIRECT. Since both the IDM 500 and CAFS are commercial database machines it is highly unlikely that we shall see any published information about the implementation experiences of either machine - in particular, a candid admission of the errors made

---

[1] By implemented, we mean that a user can sit at a terminal in his office and execute queries that are specified in a high-level query language (e.g. QUEL, SEQUEL). We view remote access to database software running on a separate machine (as in the ADABAS "database machine") as a networking problem.

and/or the success of methods used. Thus, we feel that a description of the implementation decisions that we made and a summary of our experiences will most certainly aid future database machine implementors in their work.

DIRECT has been operational since early in the spring of 1980. Presently (Summer 1981), users can interact with DIRECT through an INGRES [Ston76a] interface. DIRECT supports all the relational operators (as in INGRES) except aggregate operations and appends. Future plans for DIRECT include incorporation of aggregate operations, expansion of the hardware, and an empirical performance evaluation.

In Section 2, we present an overview of the design of DIRECT and describe the present implementation. Section 3 describes the design of the DIRECT software. In Section 4, we present the actual implementation techniques used. Finally, our conclusions and a discussion of what we have learned by implementing DIRECT is contained in Section 5.

2. DIRECT System Architecture

2.1. Background

The DIRECT project began in 1977 as a consequence of several loosely related events: experience with INGRES on a PDP 11/45, exposure to the original paper describing RAP [Ozka75a], and the acquisition of five LSI 11/03 processors through the National Science Foundation Research Equipment Program.

Our experiences with INGRES led us to feel that a back-end database machine for INGRES (or an INGRES-like DBMS) could

greatly enhance performance. While we felt that RAP could indeed serve as a back-end database machine and improve the performance of a DBMS it suffered from a number of problems. RAP.1 (the version of RAP described in [Ozka75a]) has a single instruction stream, multiple data stream (SIMD) architecture. The primary implication of this organization is that only a single instruction can be executed at a time. Thus, performance improvements over a conventional DBMS can only be achieved through the use of intra-instruction parallelism. An additional, unrelated, problem is that the entire database in RAP.1 must reside on a fixed head disk. We felt that these two factors would severely limit the performance of a large scale implementation of RAP.[2]

One of the goals of the design of DIRECT was to allow for the simultaneous execution of a number of instructions, possibly from different queries. We felt that such a capability is necessary in order for a database machine to support a high volume of transaction processing for a given time unit. Another design goal was to allow for the execution of queries on databases (or portion of databases) of arbitrary size.

2.2. Original DIRECT Architecture

As originally conceived, DIRECT consisted of six main components:

(1) A number of host processors with which users interact. Each host provides a user interface and a number of data management functions (e.g. query compilation).

---

[2] The designers of RAP recognized the limitations of RAP.1 at about the same time. Its current design can be found in [Schu79a].

(2) Some number of mass storage devices on which the database resides.

(3) A set of processors (termed query processors) responsible for the execution of relational operations on the database.

(4) A set of memory modules constructed from charge coupled ddevice (CCD) chips that are used as a shared disk cache for the query processors.

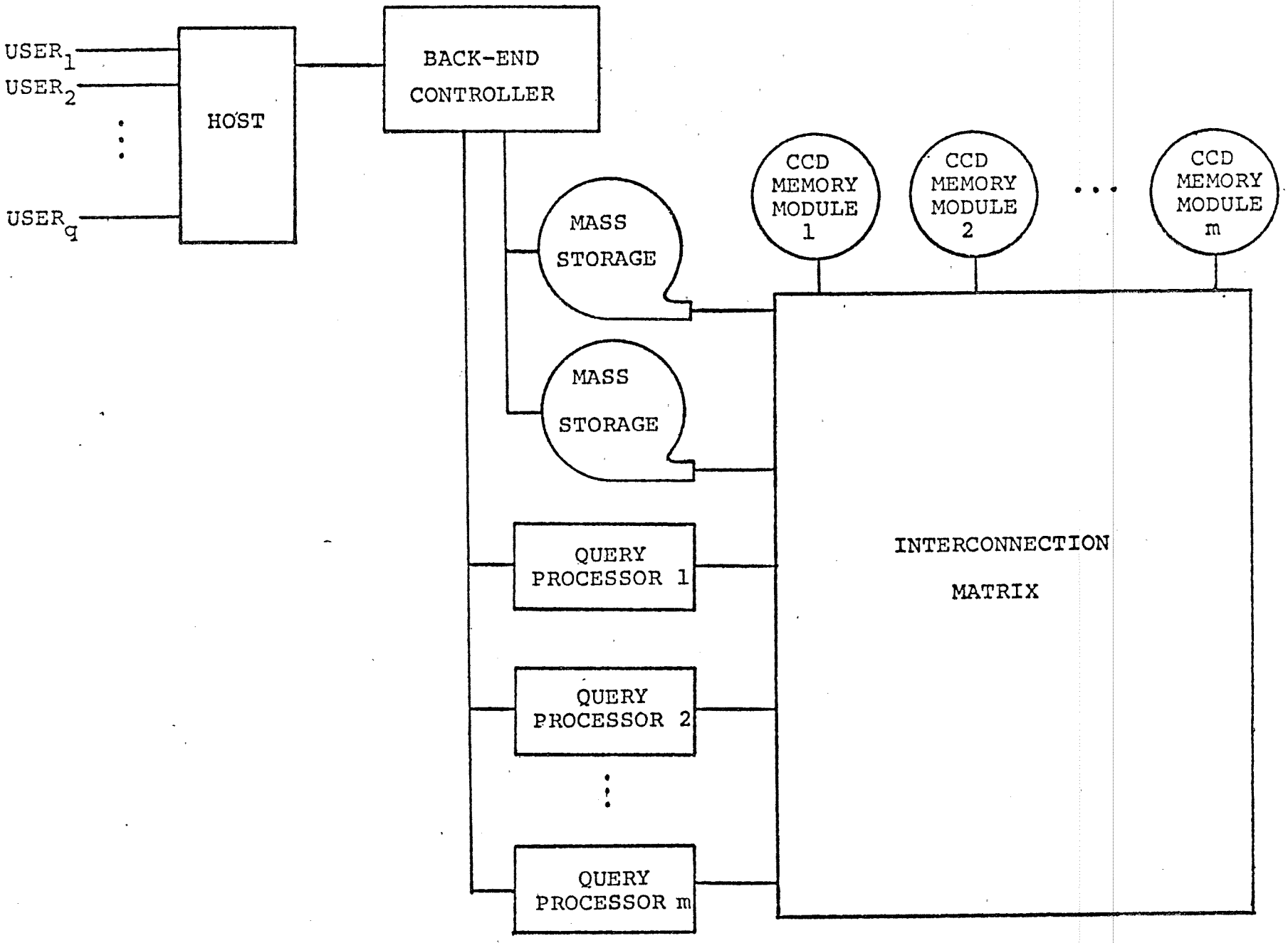
(5) A crossbar switch connecting the query processors to the CCD modules and the mass storage devices to the CCD modules.

(6) A back-end controller responsible for communication with the hosts and controlling the query processors, CCD memory modules, and mass storage devices.

A diagram of a sample DIRECT configuration is shown in Figure 1.

User queries on a host are translated into a tree format, then compiled and forwarded to the back-end controller. (An example of a query in a tree format can be seen in Figure 5). The back-end determines the "optimal" number of processors that should be assigned to the query. Processors are assigned to an instruction (an instruction corresponds to a relational operator) when it becomes enabled, i.e., when its input data exists. Thus, leaf instructions in the compiled query tree are immediately enabled. Non-leaf nodes must wait until their children have produced some or all of their output data.

DIRECT was designed to support both intra- and inter-instruction concurrency. To allow for concurrency within an instruction each relation is divided into fixed size pages. During the execution of an instruction a query processor will operate on one page of a relation at a time. To insure that each processor examines the correct subset of pages, assignment of pages to the processors is centralized and performed by the back-end. When a processor is ready to examine another page it



DIRECT SYSTEM ARCHITECTURE
Figure 1

requests the "next" page from the back-end. The back-end responds with the address of a CCD module containing a page to be examined. By maintaining various control tables in its memory the back-end can insure the correct action of a number of query processors executing the same code in parallel.

Inter-instruction concurrency also requires management by the back-end. In this case care must be taken that at the end of the concurrent execution of two or more instructions the database is in a consistent state. Since each processor must request a cache address from the back-end before it actually examines the page, consistency can be guaranteed by the back-end.

Another task for which the back-end is responsible is overseeing the transfer of data from the mass storage devices to the shared cache. This is done either in anticipation of a request or in response to one. Anticipatory paging in DIRECT can be effective because the reference strings are known in advance (a property of the algorithms used). If indices were used then the reference strings would be constructed dynamically, precluding anticipatory paging.

The shared cache consists of several CCD memory modules. Each module holds 16 Kbytes of data (the size of a page of a relation). The memory modules are connected to the query processors by a crossbar switch that has the following two properties:

(1) Two or more processors can read the same cache frame (memory module) simultaneously.

(2) Two processors can read or write two distinct cache frames simultaneously.

In DIRECT, the memory modules are the active units in a data transfer operation while the processors are passive. This means that a page stored in some cache frame can be broadcast to any number of processors. Details of the crossbar switch are in [DeWi79a].

A final comment about DIRECT is that its organization falls into the multiple instruction stream, multiple data stream (MIMD) category. Although DIRECT could operate in MIMD mode, and in past publications we have referred to it as an MIMD machine, it in fact operates as a reconfigurable multiple SIMD machine. That is, typically more than one processor is assigned to the execution of an instruction and at a given time instance more than one instruction will be active.

2.3. Current DIRECT Architecture

The current DIRECT configuration is shown in Figure 2. It consists of the following components:

(1) A PDP 11/40 running the UNIX operating system that doubles as a host processor and the back-end controller.

(2) 8 LSI 11/23 computers, each with 136 Kbytes of main memory.

(3) A multiport 1/2 Mbyte memory addressable on 512 byte boundaries. The unit of transfer is an integral number of 512 byte pages. Although this memory is constructed using 16K bit CCD chips, the logical operation of the memory is independent of the technology employed.

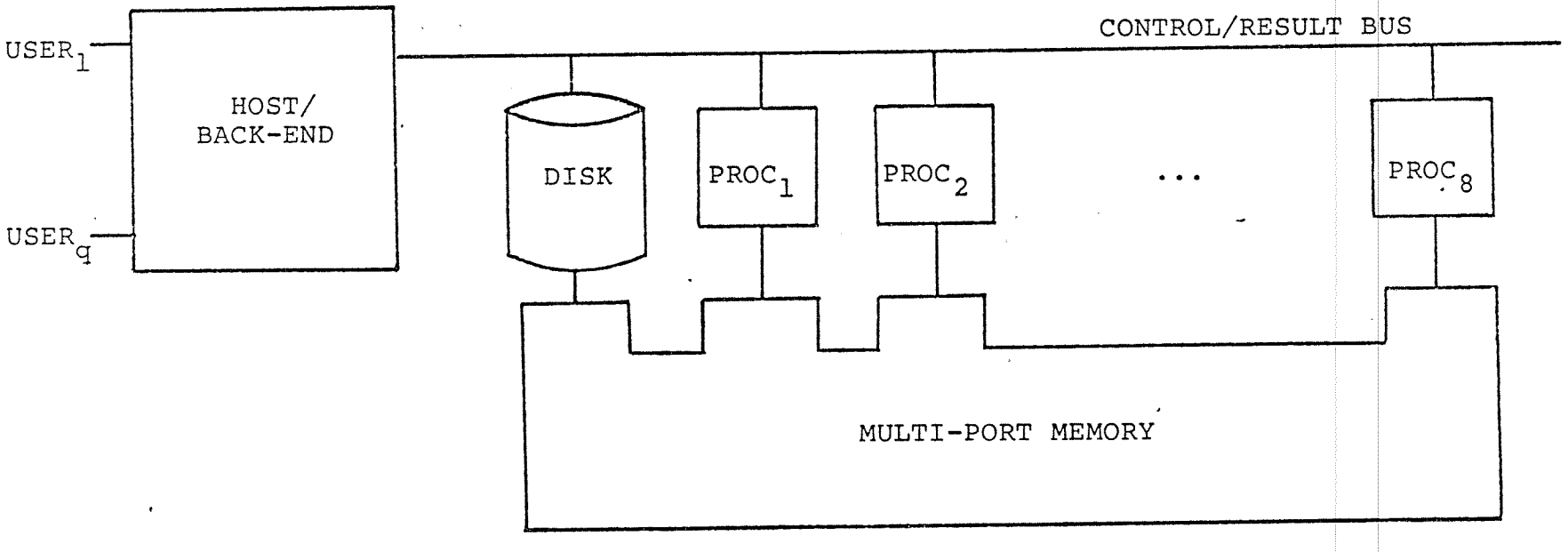
(4) A 40 Mbyte disk.

(5) A broadcast bus interconnecting the query processors, the back-end, and the disk[3] which we term the control/result bus.[4] This bus is used for the transmission of control

---

[3] Logically, the disk is connected to both the bus and the multiport memory. Physically, a processor is used.

[4] In the text we shall refer to it as the control bus.



DIRECT SYSTEM ARCHITECTURE
Figure 2

information between the query processors and the back-end; and transmission of the result relation from the disk to the back-end.

We chose to implement the crossbar switch and CCD memory modules of the original DIRECT design as a multiport memory primarily for cost reasons. However, the current configuration should attain the same performance were we to use the design that was originally proposed. Each query processor is "fooled" into thinking that the cache can be accessed by all processors in parallel. An LSI 11/23 can read data at a rate determined by its Q bus speed - about 1/2 Mbyte/second. The multiport CCD memory, on the other hand, can transfer data to and from a buffer associated with each processor at a rate of 4 Mbytes/second. The physical unit of transfer between the multiport memory and a query processor is an integral number of 512 byte pages. The logical unit of transfer is 16 Kbytes. The multiport memory, in response to a number of requests, time shares the data delivery among the processors. Since the data can be delivered 8 times as fast as a single processor can read it, even if all the processors have outstanding requests these can be satisfied without delays.

In the event that the number of processors in the configuration would grow beyond eight, each processor may spend some idle time in reading from the cache during periods of high I/O activity. However, it is not clear how many processors are required before the waiting time becomes significant. In a balanced system, at any given instant not all the processors will require access to the multiport memory.

3. Design of the DIRECT Software

In this section, we describe in some detail the host, the back-end, and the query processor process structures. While the host software is a modified version of INGRES, the back-end software was designed to process relational queries with a maximum degree of parallelism. Each query processor has some resident code (a primitive kernel) and executes instruction packets received from the back-end. We describe how the code for the query processors is generated and how a query processor operates during the execution of a code packet. Finally, we discuss the problem of placing the schema and present the solution we opted for.

3.1. The Process Structure of the DIRECT Host Software

From the beginning of the DIRECT project we assumed that we would use INGRES as the basis for implementing the user interface to DIRECT. This was done in order to minimize implementation time and maintain compatibility with the most widely used relational database system.[5]

Because of the address space limitations of the PDP 11 computer, INGRES runs as a number of processes for each active user. The actual number of processes varies between four and six depending on the particular model of PDP 11 used and whether

---

[5] Compatibility is a key issue. We feel very strongly that if database machines are to become commercially viable they must support existing DBMS software in a completely transparent manner.

INGRES query modification has been invoked.[6]

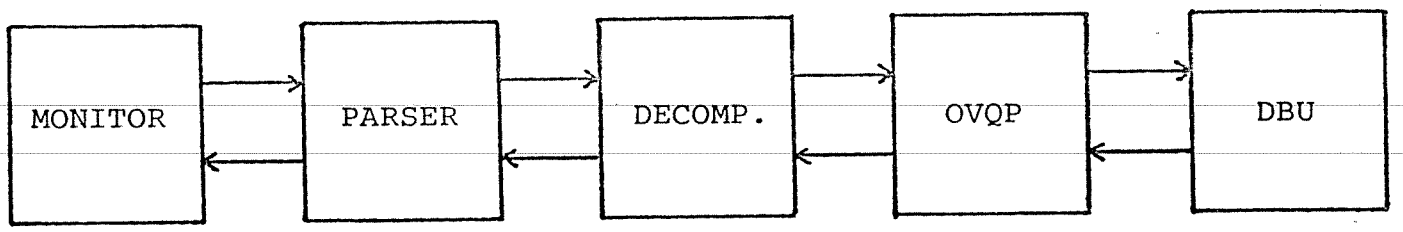
The version of INGRES which formed the basis for the DIRECT host software runs as five processes as shown in Figure 3. The monitor process is responsible for interacting with the user. Once a query has been entered by a user, it is sent by the monitor to the parser process. This process translates the query, using information from the schema, into in a binary tree format. If the query is not a utility query (e.g. print a relation) it is executed by the Decomposition and OVQP (One Variable Query Processor) processes (see [Wong76a] for more details). Finally, utility commands (to print, create, destroy, etc. relations) are executed by the DBU (Data Base Utility) process.

Since query execution is conveniently isolated as two processes, implementing the host software for DIRECT involved simply eliminating the Decomposition and OVQP processes. Figure 4 contains the process structure of our modified form of INGRES. The monitor, parser, and DBU processes perform the same functions as before. The CIDI process performs two functions. Its primary purpose is to compile INGRES queries into a form appropriate for execution by DIRECT (see Section 3.2). Its second function is to act as an interface between these four processes (for which there is an instance for every active INGRES user) and the back-end controller (which is on a separate machine).

Although we would have preferred to avoid modifying the other processes (in order to minimize the effect of each new

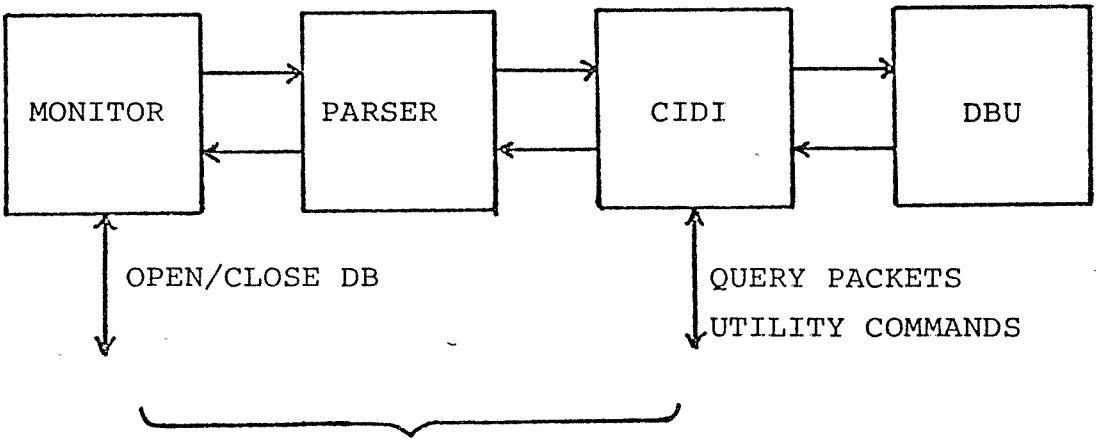
---

[6] INGRES on the VAX 11/780 runs as two processes.



INGRES PROCESS STRUCTURE

Figure 3



HOST PROCESS STRUCTURE

Figure 4

release of INGRES), we found it necessary to slightly modify the monitor and DBU processes. The monitor process was modified by the addition of two procedures to notify the back-end controller of a user opening a database upon invoking INGRES and a user closing a database upon exiting INGRES. The DBU process was slightly modified to handle differences in format and page size between INGRES and DIRECT databases. The parser process was not modified.

3.2. The Process Structure of the DIRECT Back-End Controller

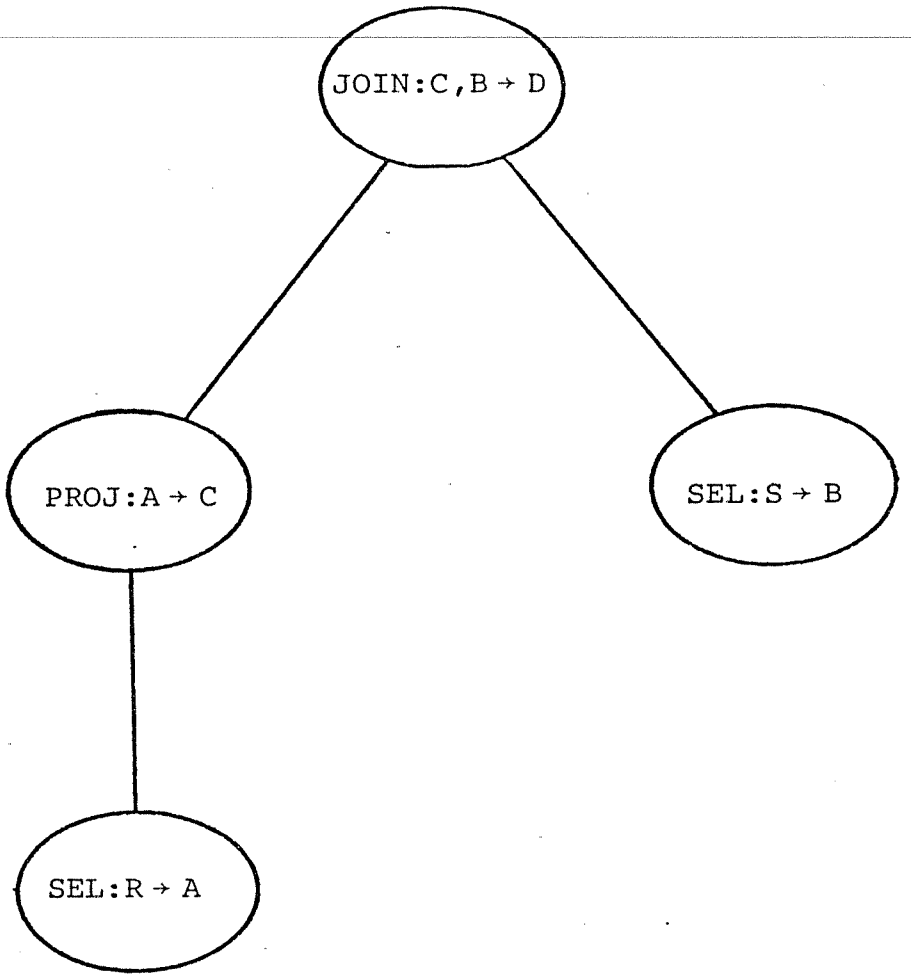
Once the functions of the host had been clearly defined, we were able to assume that the back-end machine would receive queries compiled into "packets" of instructions. Upon receiving a packet, the back-end becomes responsible for controlling the packet execution and for returning the result of the query to the host. An analysis of the services the back-end controller would have to provide revealed three main functions: catalog management, packet and instruction scheduling, and memory management. The first function requires that the back-end execute utility programs such as create and destroy a relation, or print a relation. The second function consists of controlling the execution of a packet: determining which instructions in the packet are executable and allocating a certain number of available processors to enabled instructions. This function is similar to the role of a scheduler in an operating system. The third function is the management of the shared cache. Here, the back-end processor plays the role of virtual memory manager in an operating

system.

To some extent, these functions are independent of each other and services in each of the above three categories could be provided simultaneously. Thus, there is room for some parallel activity in the back-end controller itself. We felt that by implementing each of the three functions as a separate process we would be able to measure the actual degree of parallelism possible in the back-end. Then, if additional hardware became available, we would move each process to a separate computer and exploit the parallelism. The three processes were named UTIL, PKT, and MEM respectively.

By maintaining appropriate data structures (such as relation descriptors and task descriptors) and by exchanging synchronization messages among themselves and with the query processors, the three processes are able to supervise the query processors. In order to efficiently manage the limited amount of primary memory available on the back-end, each process was allocated a "heap" from which the data structures were allocated dynamically as needed during execution of a query. Unfortunately, C (the implementation language) did not provide us with adequate support and we had to implement the heap as a fixed size array and write our own heap management routines.

To illustrate the functions of the back-end processes more clearly we shall describe in detail the execution of the query shown in Figure 5. Some of the data structures and the main messages exchanged during the execution of this query are illus-



EXAMPLE QUERY TREE

Figure 5

trated in Figure 6.[7] Each message is shown as an arrow between the box representing the sending process and the box representing the receiving process. In addition, each arrow points to the data structure to be updated upon receipt of the message. We have also shown (in the case of the GET_PAGE and NEXT_PAGE messages) the paths followed across data structures to reach the desired information.

Relations R and S, in Figure 5, are permanent relations in the database. Both relations are restricted resulting in the temporary relations A and B respectively. Relation A is projected and the result (relation C) is joined with relation B to form relation D. The query packet generated by the CIDI process (see Section 3.1) will be sent to the PKT process. PKT will first send a message to UTIL requesting the creation of the 4 temporary relations A, B, C, and D. Next, it will create a Precedence Matrix ("PPM" in Figure 6) showing the dependencies between the instructions.

For each enabled instruction (one whose inputs are available) PKT will request statistics information from UTIL, such as relation size in pages. This information is used in determining the number of processors that should be allocated to the instruction. Since the projection and join depend on output generated by the two selections, only the selections are enabled. Thus, PKT only requests information about relations R and S. Upon

---

[7] This figure is not complete. It is merely intended to sketch out some of the activities in the back-end to aid the reader in reading the text. A much used detailed figure (known as "the map") exists for internal documentation purposes.

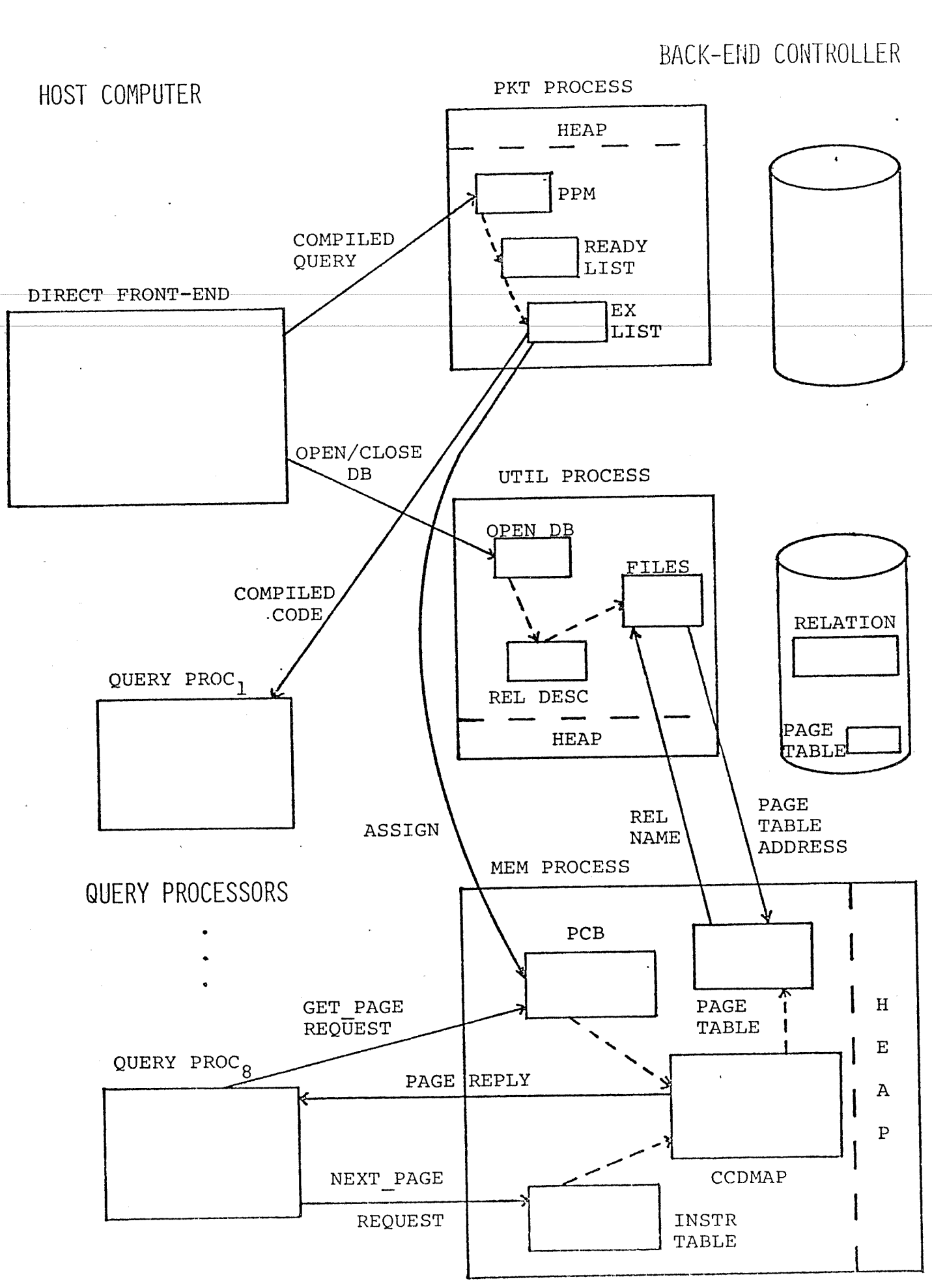

BACK-END CONTROLLER
HOST COMPUTER
PKT PROCESS
HEAP
PPM
READY LIST
EX LIST
COMPILED QUERY
DIRECT FRONT-END
OPEN/CLOSE DB
UTIL PROCESS
OPEN DB
FILES
REL DESC
HEAP
RELATION
PAGE TABLE
COMPILED CODE
QUERY PROC1
ASSIGN
REL NAME
PAGE TABLE ADDRESS
MEM PROCESS
QUERY PROCESSORS
PCB
PAGE TABLE
GET_PAGE REQUEST
QUERY PROC8
PAGE REPLY
HEAP
CCDMAP
NEXT_PAGE REQUEST
INSTR TABLE

Figure 6

receipt of the information both selections are ready to be executed. Ready instructions are placed on the READY_LIST. When a number of query processors (possibly less than the "optimal" number, and perhaps only one) become available an instruction is removed from the READY_LIST, added to the EX_LIST, and initiated.[8] PKT sends an assignment message to each allocated processor over the control bus. This message includes the compiled code to be executed (see Section 3.3). MEM is also informed by PKT each time a processor is assigned to an instruction.

MEM uses a data structure known as the processor control block ("PCB" in Figure 6) to maintain information on the activity of each processor. Additional data structures, associated with executing instructions are maintained so that an instruction execution can be monitored.

A query processor executing an instruction requests pages to be examined from MEM. Such messages are exchanged over the control bus. There are three types of page requests (described in more detail in [DeWi79a]). These are: NEXT_PAGE, GET_PAGE, and NEW_PAGE. A NEXT_PAGE request is used when a page of a relation is to be processed by a single processor, as in the selection operation. A processor requesting a NEXT_PAGE gets any page of the relation which has not yet been examined by another processor. A GET_PAGE request is used when a processor is to examine a specific page (for example one part of a stream of pages).

---

[8] There are a number of considerations used in selecting which of the two selections should be initiated first. In this paper we do not concern ourselves with this question since it is peripheral to the main subject.

Finally, a NEW_PAGE request is used when a processor is about to write a new page of a relation.

Note that MEM is truly a virtual memory manager. Pages residing in the multiport memory are grouped by classes depending on the operator that is currently using them. Classes are ranked according to priority. In the event that all the pages in the cache belong to the same class a modified LRU algorithm is employed to pick a candidate to be swapped out.

MEM responds to GET_PAGE and to NEXT_PAGE requests with the address of the page frame containing the page to be examined by the requesting processor. The processor then initiates a memory access to that frame in the multiport memory. In the event of a NEW_PAGE request, MEM must find an empty page frame in the multiport memory and send its address to the requesting processor. At times this may necessitate writing a page from one of the cache frames to disk. MEM also updates the data structures associated with the requesting processor and the instruction after each request.

When all of the relation pages have been examined MEM responds to NEXT_PAGE and GET_PAGE requests with an "end-of-relation" message. Upon receipt of this message a query processor flushes its output buffer by executing a NEW_PAGE and informs PKT that it is done. PKT, by examining a data structure associated with each instruction, can determine when an instruction has completed (i.e. all the processors assigned to the instruction have terminated). At that time PKT updates the Precedence Matrix and checks to see if any instructions are enabled as a result of

the termination of the instruction execution.

For the query of Figure 5, termination of the selection operation R->A will enable the projection. (Note that the join is not enabled until both the second selection and the projection have terminated). PKT requests statistics on relation A from UTIL. Upon receipt of the reply the projection is added to the READY_LIST. The subsequent actions taken by the back-end processes during the execution of the projection are similar to those described above. When the projection terminates the join is enabled.

After the execution of the join instruction, PKT sends a message to UTIL requesting that it print the result relation (that is, send it to the host for printing) and destroy the temporary relations created for the execution of this packet. PKT also informs MEM of the termination of the query packet and both processes destroy the data structures used to manage the packet execution.

Soon after implementation began we decided to combine MEM and UTIL into a single process. Both processes manage and use portions of the schema. In order for the two processes to run separately each would have to inform the other of any changes made. Since UTIL was expected to modify the schema relatively infrequently compared to MEM it seemed that the majority of the messages from MEM to UTIL informing it of changes to the schema would be wasted. We felt that the overhead associated with sending these messages would far offset any gains due to concurrent execution of the two processes.

3.3. Code Generation

In addressing the question of how to execute queries on the query processors, we were faced with two options: we could either have an interpreter for the query tree in each query processor and interpret instructions; or we could compile (on the host) the query tree into machine language for the query processor and execute the queries directly. The advantage of interpreting the query tree was the time and space savings in sending queries from the host through the back-end to the query processors. The query tree (as it is used by INGRES) formed a concise specification of a query and would not heavily load the communication lines. Also, writing an interpreter for the query tree would have been considerably easier than writing a code generator. The interpreter could be written in a high-level language and would be relatively insulated from changes in the query processor hardware and software.

One constraint that we faced was the amount of memory available for instructions in the query processors. We had already decided to allocate three 16 Kbyte page buffers in each query processor which left only 8 Kbytes of instruction space.[9] That space was to be divided among the interpreter and routines for the message and page I/O. Clearly it would have been a tight squeeze. Another concern was execution time. We wanted the query processors to run as fast as possible. Experience with INGRES indicated that interpreting queries does not always

[9] Our initial implementation utilized PDP 11/03 microcomputers with 56 Kbytes of memory available for program and data.

provide satisfactory response time. Thus, we chose to compile user queries into machine language for the query processor.

Having chosen the compiled approach, our next task was to decide precisely what routines should be resident in a query processor and what should be sent in a query packet. For example, because the restrict algorithm is essentially the same regardless of the relation referenced or the actual selection criterion, it would have been possible to include a restrict code "template" in the query processors. The template would consist of an outer loop which opened the relation and executed repeated NEXT_PAGE calls. An inner loop would sequence through the tuples of a page and apply the selection criteria. In this approach, all that need be sent from the host is a "compare" subroutine to apply the selection criteria specified in the query and a "copy" subroutine to move each selected tuple to the result page buffer (of course, we would also send an instruction header which identified the instruction, the relation, etc.). Similarly, there could be permanently resident join, aggregate, projection, ... templates.

We did adopt the template approach but rather than include templates for each possible instruction in all the query processors (which, again, would have required too much memory), we decided to combine the query- and relation-dependent compare and copy subroutines with the code templates on the host as part of query compilation. This saved buffer space in the query processors since we only needed to reserve space for the largest code template (rather than all of them). It also made instructions run slightly faster because having permanently resident templates

in the query processors required a subroutine call to the "compare" routine for each tuple processed. By combining the template and the compare and copy code in the host, we were able to eliminate that subroutine call.

While this approach looked promising from the beginning we were initially concerned that sending both "template" and the customized "compare" and "copy" would consume a large fraction of the available communications bandwidth. This fear was misguided since a much larger consumer of the available communications bandwidth are the NEXT_PAGE, GET_PAGE and NEW_PAGE requests. Instruction packets are relatively infrequent.

This design proved satisfactory with one exception. In hindsight, it would have been better to have the PKT process in the back-end controller (and not the CIDI process on the host) combine the generalized instruction templates with the instruction-dependent compare and copy code generated by the host immediately before initiating an instruction. This is because each code template is relatively large (e.g. the length of the code template for the join operation is approximately 300 bytes) so an entire query packet (consisting of an arbitrary number of relational algebra operations) can be arbitrarily large. Because the PKT process had only enough buffer space to hold a few query packets at a time, it was necessary to store idle packets (i.e. those for which no instruction has been initiated) on disk.

3.4. Schema Placement

Once the basic process structure of the host and back-end controller software was selected, it was necessary to decide where the schema should be placed. The host (particularly the parser, CIDI, and DBU processes) requires information about relation and attribute formats in order to parse and compile user queries. The back-end controller software also needs a limited amount of schema information in order, for example, to know how many processors to assign to an instruction. The choice we made was to distribute the schema information across both systems. This further simplified implementation of DIRECT since the INGRES schema organization was left intact and INGRES routines to look up information in the schema are heavily utilized by the code generation procedures of the CIDI process. However, care had to be exercised to insure that the duplicate schema information was kept in a consistent state. By using that INGRES software which prevents a user from destroying a relation being used by another, we found that keeping the duplicated schema information in a consistent state was straightforward.

3.5. Query Processor Operation

In Figure 7, a map of a query processor's memory is shown. The query processor monitor basically loops, waiting for an instruction to arrive from the back-end controller. Instructions are read into the instruction buffer by the monitor which then starts execution by jumping into the instruction buffer as if it were a subroutine. The instruction invokes the page and message

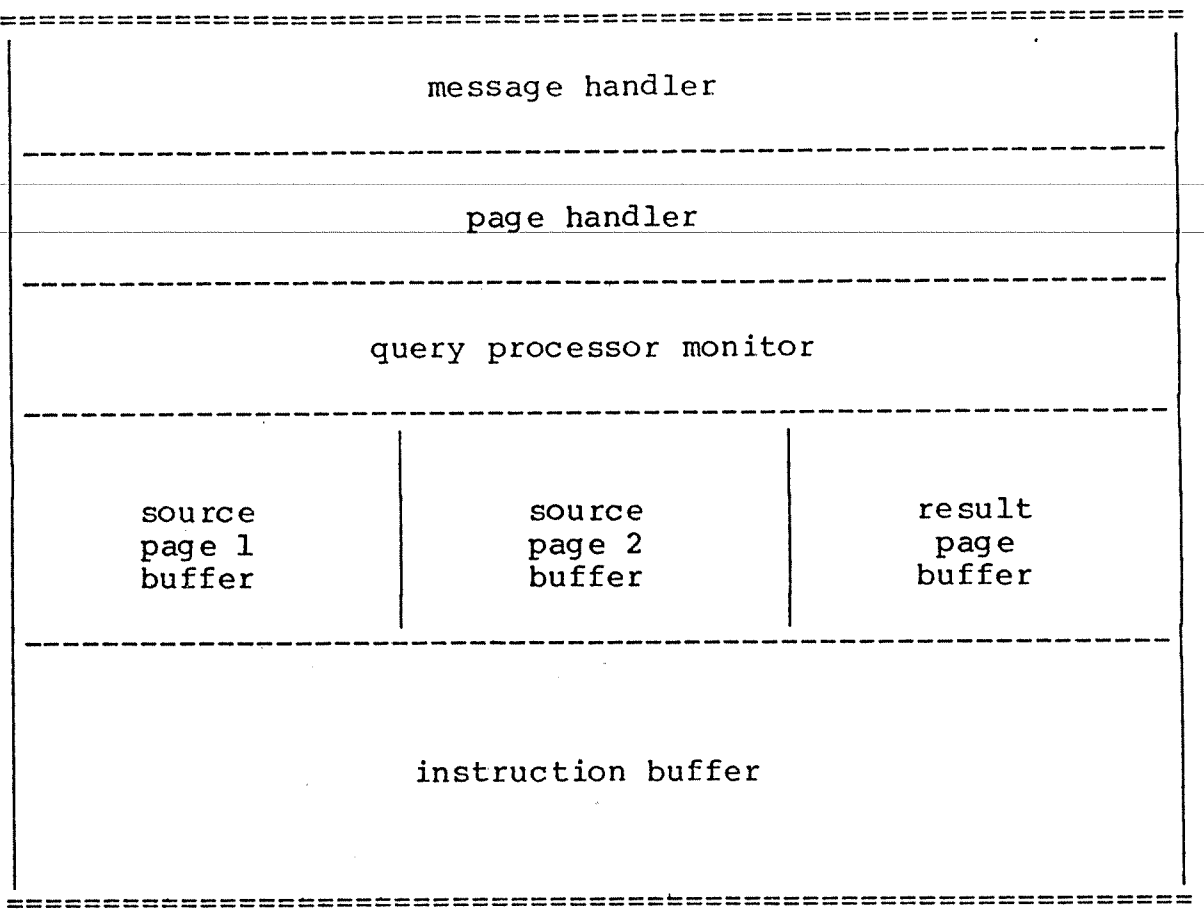


Figure 7
Layout of Query Processor Memory

handlers as necessary and when finished, returns control to the monitor.

4. Implementation Issues

4.1. Interprocess Communication

Our design required an interprocess communication (IPC) facility that would enable processes on the host computer, the back-end controller, and the query processors to communicate with each other. On the host, each CIDI process ("representing" a

user query) must send query packets to PKT, and utility messages (e.g. create a relation) to UTIL. It must also wait for replies from these processes indicating the status of the operation and possibly a result. On the back-end, PKT, MEM and UTIL must communicate with each other. PKT and UTIL must also communicate with an unspecified and varying number of CIDI processes on the host. Finally, PKT and MEM communicate with the query processors.

In Version 6 UNIX, pipes are used for IPC. One feature of pipes is that processes communicating over a pipe must be related (i.e. a parent and a child). Although there was no problem in making PKT, MEM, and UTIL siblings there was no easy way of doing that for the CIDI processes. Each CIDI process is part of an DIRECT process chain that is formed at the time a user invokes DIRECT and destroyed at the time the user exits from DIRECT. Pipes, therefore, proved unsuitable for communication between the back-end processes and the CIDI processes on the host.

One alternative to pipes (in Version 6 UNIX) is ports [Zuck77a]. Ports are similar to pipes in their capabilities and usage.[10] The main difference between a port and a pipe is that two or more unrelated processes can communicate over a port. Each port is "owned" by a single process which is the only process that can read from it.[11] Any number of other processes

---

[10] See Section 5 for a discussion of some of their shortcomings.

[11] Actually, since in UNIX children processes inherit all their parents property, a child of a port owner could also read from that port. In practice, chaos would result were a program-

can write to the port. We decided to use ports rather than pipes because they offered us the possibility of writing a single communication subsystem that could be used by all the processes on all the machines. In retrospect, this was a good decision because debugging the communication code proved to be a lengthy and cumbersome task.

4.2. Implementation Technique

We began implementation before all the hardware was available: there were problems with the LSI 11/03 microcomputers that served as the query processors and the multiport memory was still under construction. Implementation of the back-end processes on the one hand and the host CIDI process on the other hand proceeded independently and in parallel.

Debugging of the DIRECT software occurred in three steps. In each step we configured a new system. These were known as FQRYP, QRYP, and CQRYP. In FQRYP both the query processors and the multiport memory were "faked". Once DIRECT was operational under FQRYP we moved to QRYP in which actual processors were used for the query processors but the multiport memory was still faked. In the final stage, CQRYP, the processors as well as the multiport memory were used.

To fake a query processor we implemented it as a separate process which included the resident routines that would otherwise be present in the query processor's memory.[12] Assignment messages

---

mer to use this capability.

[12] Actually, these routines were slightly different - they included more debugging features.

from PKT (those that included code) were read into an array and were executed by jumping into a particular address in that array.

To fake the multiport memory, we allocated an array of vectors in MEM. Each vector consisted of 512 bytes and represented a page frame (the page size was reduced from 16 Kbytes to 512 bytes). The number of vectors corresponded to the number of page frames. Pages were read from the disk into the array. A query processor still sent a page request to MEM and received a reply containing a page frame number. Then, the processor initiated a data transfer from the fake multiport memory by requesting the contents of a specific page frame from MEM. MEM responded with another message that included the desired page. Although this procedure was cumbersome and added unnecessary communication overhead, it made the move to CQRYP easier than it would have been had we combined the reply to the request message with the data transfer message.

The three systems still run. We recently used FQRYP to test code installed to perform update operations.

4.3. Implementation Time Frame

Our work on modifying INGRES (restructuring the process structure and writing the code generator) required a little less than a man-year. We spent an intensive three month period writing and debugging FQRYP, altogether another man-year. The move from FQRYP to QRYP took 3 days - we had to redo some of the message handling code. While approximately 5 days of effort were required for the move from QRYP to CQRYP, the actual time period

was about 4 months during which several hardware bugs were discovered (and fixed).

It should be noted that one of the reasons the move from FQRYP to QRYP took such a short time is that the software for downloading the LSI microcomputers and using the hardware devices for interprocessor communication had already been developed by another group using the processors for research in distributed operating systems [Solo79a].

To recap, use of emulation as a technique for implementing the system simplified debugging and allowed us to develop different parts of the system independently and in parallel. Also, the code written for the emulated system was usable on the hardware with trivial changes. We still use FQRYP every time a new addition is made to the system. We feel that this implementation technique was very successful because it enabled us to work in an environment where debugging was relatively easy and because no work had to be duplicated in moving to the hardware. It should be noted that one reason we were able to follow this path is that all the computers used were of the same family.

5. What Have We Learned?

In this section we present what we feel to be the important lessons of our work on DIRECT in the past four years. We briefly discuss the lessons gained from a simulation of DIRECT, undertaken early on in the project. We then describe some of the problems we encountered during the implementation. This is followed with an enumeration of the "valuable" lessons we have

learned. We close with a list of programming tools that we would have liked to have had and a list of future activities.

5.1. Simulation

After the initial design of DIRECT was completed we undertook to implement a detailed simulation. Concurrently, we analyzed parallel algorithms for all the relational operators for a DIRECT-like architecture. Ideally, both of these efforts should have taken place before we began work on the implementation. However, various "economic" factors forced us to commence implementing the system before sufficient ground work had been done.

The primary purpose for implementing the simulation was to study a number of different strategies for allocating processors to tasks. What began as a small limited-scale project ended up as a major effort. Once the simulation was running we were indeed able to select a "best" processor allocation strategy (see [Bora81a]). We also found out that DIRECT (as described in [DeWi79a] suffers from a number of problems.

Most important of these is the high cost of controlling the execution of a query. Our simulation showed that except for small system configurations (up to 20 processors) the execution time of various query mixes was dominated by the time required by the back-end to process the control messages (e.g. NEXT_PAGE). Another problem is DIRECT's poor performance when executing selection-only queries. A page of the relation to be restricted must be moved from the mass storage device into the CCD memory

and from there into a processor's memory before the selection operation can be applied. Since a selection requires a single scan of the data most of the execution time is spent doing "useless" work (i.e. the I/O transfers). Yet another problem is that all data transfers between processors are performed at the page level through page frames in the CCD. Although this approach minimizes transfer overhead by grouping data in large units, it results in a significant amount of page fragmentation. In particular, we found that processors executing some operation will, in general, not output full pages. Thus, portions of the CCD memory remain unused.

The simulation proved very helpful during the actual implementation in a number of ways. For example, because of the fine level of detail simulated we had a clear idea of the different functions the back-end had to perform. This information was used in determining the process structure in the back-end. We also learned what data structures were required in order to implement the needed functions. Finally, the simulation provided us with the capability of testing different memory management algorithms to be used by the MEM process.

5.2. Implementation Hassles

DIRECT was implemented using the C language on the UNIX operating system. At the outset, we did not appreciate the difficulties of writing a large system. The lack of structure in C (as compared with Pascal) exacerbated the problem. Many simple coding bugs were found only through time-consuming trial-and-

error search or serendipity. A related problem was that our development machine, a PDP 11/40, had insufficient memory to run UNIX Version 7. Thus, we were unable to take advantage of the new debugging and development tools available under this system.

Our use of ports for interprocessor communication resulted in a number of unanticipated (and serious) problems. A process attempting to read an empty port is blocked until the port is filled. This would have been acceptable had there been a way to detect the presence of a newly arrived message. But there was not. Similarly, a process attempting to write to a full port is blocked until sufficient room for the message exists. Since there were numerous processes writing to each port (for example, an unspecified number of CIDI processes, MEM, and all the query processors write to the PKT port), blocking could result as a side-effect of writing to a full port. We were fearful that this could lead to deadlock. We therefore placed constraints on the number of outstanding messages that a process could have at any given time. While we convinced ourselves that deadlock will be prevented, this was achieved at a possible performance loss since each back-end process could not attend to a task until receiving a reply to an outstanding message.

An additional problem with ports is that large messages may be split into smaller packets. The intent of this feature is to prevent a single writer from monopolizing a port. Unfortunately, message splitting is not under control of the application and it is difficult to predict when it will occur. The result then, is that ports cannot really be treated as stream I/O devices. This

greatly complicated the implementation of the communication code and was a source of numerous bugs.

One of the problems we suffered from was lack of coordination among ourselves during coding. Initially, each person worked individually on a separate process. However, later we began to combine our code. A typical example of the difficulties we encountered at that stage is when we discovered that in the MEM process the named constants TRUE, IN, and YES were used synonymously but defined differently.

5.3. Valuable Lessons

We made two important decisions at the outset of the code design phase with which we are happy to this day: to use as much of INGRES as possible and to work in an emulated environment. We believe that the short section on implementation time frame (Section 4.3) speaks for the success of the emulation method.

The decision to use the INGRES user interface saved several person years of effort; effort that would have been expended on rather mundane aspects of DIRECT, at that. Thus, we had a head start on our implementation by using INGRES. Adapting INGRES to our needs was not trivial, however. The system was fairly well structured but it was commented with restraint (the DIRECT code is no better in that respect). For example, the INGRES code contained much debug print which was enabled with run-time flags. But, it was never clear to us how to set those flags. And something as simple as a global symbol table and cross reference index would have been very useful. Much time was spent flipping

through pages looking for a particular subroutine. In short, for large programs such as INGRES, modularity and good structure must be supplemented with auxiliary internal documentation.

DIRECT now supports most of the data manipulation capabilities provided by INGRES. We will soon begin to measure its performance. Instrumentation of the system will most likely yield some information that will lead to performance improvements. Unfortunately, the task of instrumenting DIRECT will not be easy because at coding time we did not put hooks into the code to enable us to do this. There are certain places that one can be sure to find performance problems (e.g. access to frequently used data structures). Nonetheless we regret not having had the foresight to organize the code better for this task.

A related problem is that of a test database. During our debugging runs we used a supplier-parts database. This database consists of 9 relations, the largest of which has 1200 bytes. The DIRECT page size was reduced from 16 Kbytes to 512 bytes to test execution of an instruction by more than a single processor. Although it was necessary to use a small database for debugging, we now we need a "large" database for testing performance. To our regret we have not been able to procure a database much larger than the supplier-parts database we have been using. One long-range goal is to compare the performance of DIRECT and INGRES in processing queries on large databases.

Our experiences with the customized hardware (the multiport memory and the various interfacing equipment) have led us to feel that in a university environment one should attempt to minimize

the amount of custom designed hardware. Such hardware should be designed and constructed only after all other alternatives have been exhausted. It may be the case that a few years from now, when sophisticated VLSI design tools become available in universities that custom designed hardware may be easier to get.

One observation that we can make about the performance of DIRECT at this time is that messages between machines require between 10 and 15 milliseconds. Other distributed systems researchers have found this cost to be approximately the same. The majority of this time is spent in the software. Although the cost of message handling in software can doubtlessly be reduced (through the use of microcode or even providing some hardware primitives) the cost of processing a message is likely to remain high. This is a significant problem, and one that seems to be steadfastly ignored by designers of distributed systems (including multiprocessor database machines).

We conclude with a description of those tools that would have aided system development. The ability to share data structures between processes would have simplified the implementation of the MEM, UTIL, and PKT processes and would have reduced the number of interprocess messages necessary to exchange information about the status of an executing instruction, for example. The most complex code in DIRECT (which has already been rewritten three times) is the code to handle interprocess messages. The complexity arises mainly because the Rand port mechanism splits messages at arbitrary places. A more flexible IPC facility which permits the process to specify never to split messages would have

significantly simplified this section of code. Another desired feature is the ability to detect the presence of an incoming message rather than blocking when a read is attempted on an empty port. Finally, a more structured programming language with associated development system would have certainly facilitated the debugging task.

5.4. Future Plans

Although DIRECT is operational, it is by no means dead as a research project. Currently planned activities for the future include the incorporation of aggregates (including aggregate functions); instrumenting the code and making it more efficient; obtaining a "large" database and comparing the performance of DIRECT to that of INGRES; incorporating concurrency control and recovery; moving the back-end to a VAX (this entails switching from Version 6 UNIX to Version 7 and redoing the front-end since the INGRES process structure for VAX UNIX is different); and moving MEM/UTIL and PKT to separate processors.

6. Acknowledgements

We would like to finish by thanking the INGRES group, in particular Bob Epstein and Eric Allman, for their help in advising us about modifying INGRES. Michael Stonebraker and Gene Wong were kind to allow various INGRES people to spend time helping us.

However, despite their counsel, two routines, "copy_to" and "copy_from", still remain a complete mystery to us. These routines are used to copy UNIX files into relations and vice versa.

Initially we attempted to modify these routines so that they would directly convert UNIX files to and from the DIRECT relation format. However, the code was so obtuse, we gave up and wrote a procedure that first converts the file to the INGRES relation format (which is well documented) and then applies a transformation to the DIRECT relation format. Perhaps not elegant, but it works.

7. References

[Babb79a] Babb, E., "Implementing a Relational Database by Means of Specialized Hardware," ACM TODS 4, 1, (March 1979).

[Bora80a] Boral, H., D.J. DeWitt, D. Friedland, and W.K. Wilkinson, "Parallel Algorithms for the Execution of Relational Database Operations," ACM TODS, (Submitted October 1980).

[Bora81a] Boral, H. and D.J. DeWitt, "Processor Allocation Strategies for Multiprocessor Database Machines," ACM TODS 6, (June 1981).

[DeWi79a] DeWitt, D.J., "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems," IEEE Transactions on Computers c-28, 6, (June 1979).

[DeWi79b] DeWitt, D.J., "Query Execution in DIRECT," Proceedings of the ACM SIGMOD 1979 International Conference of Management of Data, (May 1979).

[Epst80a] Epstein, R. and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," Proc NCC 49, AFIPS, (1980).

[Hawt81a] Hawthorn, P. and D.J. DeWitt, "Performance Evaluation of Database Machines," IEEE Transactions on Software Engineering, (To Appear 1981).

[Hawt81b] Hawthorn, P., "The Effect of the Target Applications on the Design of Database Machines," Proc of the ACM SIGMOD 1981 International Conference of Management of Data, (May 1981).

[Ozka75a] Ozkarahan, E.A., S.A. Schuster, and K.C. Smith, "RAP - An Associative Processor for Data Base Management," Proc NCC 45, AFIPS Press, (1975).

[Schu79a] Schuster, S.A., H.B. Nguyen, E.A. Ozkarahan, and K.C. Smith, "RAP.2 - An Associative Processor for Databases and Its Applications," IEEE Transactions on Computers c-28, 6, (June 1979).

[Solo79a] Solomon, M. and R. Finkel, "The Roscoe distributed operating system," Proceedings 7th Symposium on Operating Systems Principles, pp. 108-114 (December 1979).

[Ston76a] Stonebraker, M.R., E. Wong, and P. Kreps, "The Design and Implementation of INGRES," ACM TODS 1, 3, (September 1976).

[Wong76a] Wong, E. and K. Youssefi, "Decomposition - A Strategy for Query Processing," ACM TODS 1, 3, (September 1976).

[Zuck77a] Zucker, S., "Interprocess Communication Extensions for the UNIX Operating System: II. Implementation," R-2064/2-AF, RAND (June 1977).