MUFFIN: A DISTRIBUTED DATABASE MACHINE

by

M. Stonebraker

Memorandum No. UCB/ERL M79/28

1 May 1979

MUFFIN: A DISTRIBUTED DATA BASE MACHINE

by

Michael Stonebraker

Memorandum No. UCB/ERL 79/28

1 May, 1979

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MUFFIN: A DISTRIBUTED DATA BASE MACHINE

by

Michael Stonebraker

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

UNIVERSITY OF CALIFORNIA

BERKELEY, CA.

ABSTRACT

This paper discusses the design of MUFFIN (MUltiple Fast or Faster INgres), a distributed data base machine being developed at Berkeley. The basic design goals are higher transaction rates than achievable through conventional means, resiliency to failures and support for a wide variety of data bases in terms of size and complexity of transactions. The design exploits the possibilities of parallelism among large units of work, variable size RAM caches, no concept of "GOD" and streamlining of modules through specialization of function.

The design is contrasted with conventional distributed data base systems and with other proposals for multiprocessor data base machines.

Research sponsored by the Office of Naval Research Contract N0039-78-C-0013 and the National Science Foundation Grant MCS75-03839-A01.

I INTRODUCTION

This paper discusses MUFFIN (MUltiple Fast or Faster INgres), a data base machine under development at Berkeley. It differs substantially from an earlier proposal [STON78] and bears little resemblance to other so-called data base machines [SCHU78, DEWI78, BANE78, LIN 76, COPE73]. The goals of MUFFIN are the following:

1) higher transaction rates (by at least a factor of 10) than those achievable by one-machine conventional architectures.

2) support for geographically remote data in a transparent way.

3) support for data bases in the 1 mbyte to 100,000 mbyte range.

4) modular expansion, i.e. the ability to expand transaction processing capability and/or data base size in a reasonable way.

5) resiliency, i.e. the ability to survive without crashing and without data loss or data u availability at least all single component hardware failures.

The next section discusses the architecture of MUFFIN and indicates some of the major design decisions that have been made. After that, we indicate several alternative architectures and our reasons for rejecting them. Then in Section 4 we discuss performance considerations and indicate the environment in which the intitial MUFFIN prototype will run. We conclude by indicating the implementation status of the prototype.

II THE ARCHITECTURE OF MUFFIN

2.1 Overview

MUFFIN consists of two types of processing nodes A-cells (application program cells) and D-cells (data base nodes). These cells are connected into "pods" using a high speed bus or local network [CLAR78]. There is no limit on the number or type of cells in a pod. Pods are connected to other pods through gateways and lower speed communication links. Figure 1 indicates an example of this architecture having two pods of respectively two and four cells. It should be clearly noted that other types of nodes may in fact be physically connected to a structure such as Figure 1. However, data base software only knows about the above two kinds of cells.

We first discuss D-cells and A-cells; then we consider the bus. Lastly, we treat the software environment. The lower speed communication link is deferred to Section V.

2.2 D-cells

Figure 2 indicates the nature of a D-cell. It consists of a mandatory part containing a bus interface, a rather conventional processor, random access memory along with an optional disk system.

The following are the key features of a D-cell.

1) The processor is totally dedicated.

The only executable process on a D-cell is the run-time data base system code. The only interface to the outside world is through the bus. Moreover, there are no other peripherals connected to a D-cell.

2) There is no information managed by a D-cell except a portion of a data base.

This data base may reside in RAM if the optional disk is not present. Otherwise, RAM acts as a cache for the data base. The size of this cache is an important (application dependent) performance parameter.

3) The protocol understood by a D-cell is highly stylized.

The D-cell processor is capable of accepting ONLY messages containing data nanipulation language commands (DML) in QUEL [HELD75] and responding with appropriate data and/or completion information.

4) There is no general purpose operating system present in a D-cell.

At best, a very lean collection of needed utilities will be included.

5) A variety of disk systems are possible.

The disk system is not constrained except that there be one disk controller. Conceivably, the storage medium could be a floppy disk.

6) The processor can be quite ordinary.

The instruction set of the processor should be oriented toward supporting ONLY the run-time data base system. In this context, there is little need for memory management or virtual memory or hardware protection. By and large the instructions supported should be conventional with perhaps a "smart" string compare command included.

2.3 A-cells

A-cells are conceived to be either a "smart" terminal or a time-sharing system. In either case they will run an operating system such as UNIX [RITC75] and some complement of peripherals. The important characteristics of an A-cell are the following:

1) The application program from which data base requests ultimately originate must run on an A-cell.

2) Some portion of a data base may reside optionally on an A-cell.

2.4 The Bus

The cells in a pod are connected together using a bus or local network. Each cell has a bus interface (say the LNI [CLAR78], or CHAOS [CLAR73], or maybe even a UNIBUS flavor of interface). Software in an A-cell must be able to originate messages to all cells, broadcast messages to subsets of the cells (if that feature is not supported in the hardware), appropriately synchronize responses, and multiplex the bus among multiple sending processes. As such, the software overhead in an A-cell to send a message may be non-trivial.

A D-cell, on the other hand, never originates messages. It simply accepts messages from the bus, executes them and returns an answer. As such, the overhead in a D-cell should be much lower.

If the overhead to send and receive messages becomes intollerable, then perhaps the communications code can be run in a separate processor sitting between the bus interface and the main processor. This tactic has been employed by the COOLS system of the Bank of America.

2.5 The Software Environment

The collection of processors described above runs the distributed INGRES software system [EPST78, STON78b]. Hence, from a software point of view MUFFIN looks exactly like a distributed data base system (such as SDD-1 [ROTH77]).

Distributed INGRES supports a user from an application program manipulating a data base in QUEL that physically resides on several processors as if it was resident on his local cell. Such QUEL commands are processed by a sequence of local transactions at various sites intersperced with data movement among sites. The processing algorithms for distributed INGRES are given in [EPST78]. Here, we only note that processing on behalf of a transaction can be done in parallel at various sites (so-called intra-query parallelism [DEWI78]), broadcast is a very useful feature, and multiple commands from different application programs can be executed concurrenctly (so called inter-query parallelism [DEWI78]).

Figure 3 shows the run-time software environment which must be supported. Here, an application program (or terminal monitor) talks to an end user at a terminal and generates data base transactions. These are passed to a "master INGRES" process which we expect will run in the same A-cell as the application program. The master INGRES coordinates the execution of a transaction by making calls on "slave INGRESs" which run at each site where processing on behalf of the transaction takes place.

These calls are either to run a local QUEL command or to move data to some other subcollection of sites. A local query is transmitted in parsed form. Consequently, slave INGRES code contains only QUEL processing code and some of the utilities. On the other hand, a master INGRES must perform parsing, query modification for support of views, integrity constraints and protection [STON75], and have all the utilities.

We can now discuss further details of the D-cell software.

2.6 D-cell Software

The INGRES algorithms for query processing have been discussed elsewhere [WONG76]. In the interests of higher performance in a dedicated environment, we are making the following changes to the one-machine algorithms.

1) There is no file system in the conventional operating system sense. Rather there is a relation storage system whose details are discussed in the next section.

2) There is no physical protection of data objects. Clearly, the only process able to access data is the run-time DBMS. Query modification is all the protection that is needed, and it runs at an A-cell.

3) There is no operating system per se. The slave INGRES simply reads commands from the bus and executes them. Presumably simple multi-tasking is done, although the case can be made against it [STON73].

4) There is no memory management in the conventional sense. The slave INGRES code is entirely main memory resident; the rest of the RAM is a buffer pool.

5) The buffer pool is managed by the DBMS which always knows, for each access to the buffer pool, which of the following four situations it is in.

a) This is not the last reference to the current page; it should remain in the cache.

b) This is the last reference to the current page for now, but it is likely to be re-referenced soon (e.g. inserts to an indexed access method when there are overflow pages present, inserts to the audit trail, etc.). The page should remain in the cache.

c) This is the last reference to the current page, but somebody else may reference it soon (e.g. directory pages for an indexed access method, system catalog pages, etc.). The page should remain in the cache if there is room.

d) This is the last reference to the current page, and it is not likely to be referenced again (ordinary data pages). The page should be deleted from the cache.

It is clear that treating the four classes of pages differently in a replacement algorithm will result in a composite scheme which will substantially outperform a simple one-class LRU scheme such as implemented in UNIX. We plan to implement just such a four class replacement strategy.

6) The DBMS knows exactly when it is viable to prefetch a page and additionally knows EXACTLY which one to prefetch.

Again, by implementing exactly the prefetch policy which the DBMS knows is best, we expect to substantially outperform typical operating system prefetch strategies. For example, UNIX blindly prefetches the next sequential page when it detects sequential access. This is only one case of several where the next page is predictably known in advance. A DBMS algorithm can take advantage of all cases.

7) Some commands involve examining a large collection of records. In this case, the current INGRES strategy of interpretive execution is wasteful of CPU resources [STON79, HAWT79]. However, if only one or two records will be accessed, there is little penalty for an interpreter. The run-time DBMS knows when a command may involve examining many records, and will be modified to incrementally compile on-the-fly a procedure for iterating through a relation selecting qualifying tuples (the one variable query processor portion of INGRES). Simple commands will continue to be interpreted.

This strategy represents far less cost than writing a compiler from scratch (as in System R [LORI77]). Moreover, we expect to suffer only a small (if any) penalty for simple commands compared to compilation. Complex transactions may well run faster, because we are interpreting access path selection and order of joining relations. Hence, such important decisions can be made on-the-fly based on accurate information on sizes of intermediate scratch relations.

8) The query processing algorithms appear to work well except in the case of certain equi-joins between relations. Here, a merge-sort tactic appears to be useful [BLAS77, SELI79]. Hence, we will add this tactic to the query processing strategy.

However, it should be clearly noted that a merge-sort tactic will have exactly as many page faults as a tactic which arranges to sort both relations in question and then do tuple substitution as in [WONG76]. Hence, no I/O time will be saved; rather we expect to economize on CPU resources using this tactic.

2.7 The Relation Storage System

A relation is stored in a collection of extents (16 or less), each of which is a collection of physically contiguous 4096 byte data blocks. Available extents are kept on a free list with a header in a known location (say in block 1). Block 0 will presumably be required for booting the D-cell. Lastly, the structure of the system catalogs may have to be stored at a fixed place on the disk and used during boots. The other alternative is to "hard wire" it into the D-cell code.

The system calalogs are identical to the ones for distributed INGRES with two exceptions. First, the "relation" relation (which has one row in it for each relation on the disk system) must have 32 fields added to it. These are:

```
 extent_0_start_address
 extent_0_length
    .
    .
    .
 extent_15_start_address
 extent_15_length
```

Second, relations are currently grouped together into data bases. Since there will be only one relation relation (instead of several in the current system), we must distinguish which data base a relation belongs to. This will be done by prepending to each relation name the data base name to which it belongs.

Notice that there is no need for any protection or for a tree structured collection of objects, as in UNIX [RITC75].

Lastly, we plan to have a single access method which will be a dynamic directory structure for keyed files (as in B-trees) with secondary indices. The reason to have only one is to save space in the slave INGRES code so it has some chance at fitting in 128K of address space (see Section VI). The reasons for B-trees has been discussed in [STON79] and will not be repeated here. This access method will be aware of cylinder boundaries and ensure that directory pages are put on the same cylinder as pages which they point to. This will save one seek during each random access to a data page.

2.8 Addressing in a D-cell

Main memory in a D-cell can look as shown in Figure 4. Note that there is little (if any) advantage to any virtual memory scheme since memory management is only involved in manipulating the buffer pool. Also, there is little advantage to a one level store (such as implemented in the IBM System/38). Using a virtual memory approach, I/O is done by binding a file into the address space of a process and then handling page faults. Notice that the information on extents contained in the relation relation is a more compact representation of disk information than a page map would be. Hence, space is economized by a "relation descriptor". Also, fancy buffer management (such as discussed in Section 2.6) is difficult to implement in a virtual memory environment.

Consequently, we expect addressing to be conventional with no hardware address modification.

III OTHER ALTERNATIVES

In this section we suggest some of the architectural choices which we considered and dismissed.

3.1 Identical Processors

Several other proposals (for example X-TREE [DESP78] and CM* [FULL77]) have suggested using identical hardware nodes connected into some structure.

Although we understand the reasoning behind such proposals, we feel that it is impractical in our environment. There appears to be a large cost savings to providing only needed functions in a D-cell and not all the functions which an A-cell must have. Hence, A-cells and D-cells are expected to have different instruction sets.

In addition, we expect some A-cells will be smart terminals and some will be general purpose main frames with a complement of peripherals. Obviously, the same CPU will not serve both situations. It should be noted that we can conceive of applications consisting of only smart terminals and no D-cells or main frame A-cells. In this case we have an "ultimately" distributed data base with data under the keyboard of each operator interacting with it. Consequently, restricting A-cells to be main frames seems inappropriate.

3.2 Non Bus Connections

Many other systems do not have a common bus for communication (e.g. DIRECT [DEWI78], CM* [FULL77], RAP [SCHU78], X-TREE [DESP78]) In our application we expect the benefit of broadcast to be dramatic. This obviously suggests a bus scheme.

3.3 Hierarchy of Processors

Several systems have proposed a hierarchy of processors (e.g. X-TREE [DESP78]). If such a hierarchy is a reasonable implementation of a bus, then MUFFIN cells could simply be the leaves of such a tree. However, if data base processing is to be performed at non leaf nodes, then two serious problems result.

a) resiliency

One obvious tactic is to use higher level nodes for control information (e.g. concurrency control or crash recovery). This, in essence, implies that concurrency control and crash recovery are at least partly centralized. Although crash recovery in a distributed data base appears to be poorly understood (at least in the presence of multiple copies of objects), it appears that "godless" schemes (e.g. [STON78b]) will substantially outperform schemes with "god" (e.g. [MENA78]).

This conclusion suggests non hierarchical architectures.

b) bandwidth and work distribution

Suppose each leaf cell in a hierarchy has an equal size fragment of each of two relations which are to be joined together. Further suppose the natural hierarchical algorithm is used whereby each cell passes fragments upward if they must be further examined. Moreover, each cell does whatever portion of the join it can do that has not been done by a lower processor.

A simple calculation will verify that 50 percent of the total work using this algorithm must be done by the root node. Moreover, every single fragment must be examined by the root; consequently, it must have enormous bandwidth.

No scheme occurs to the author which does not create a bottleneck at the root in a wide variety of situations. The only way to avoid overloading the root is to generate a data base design which minimizes the number of fragments of a relation. This clearly reduces possible intra-query parallelism.

3.4 Processors on the disk head

Various associative disk proposals have been made over the last ten years. These include [BANE78] and [SLOT70]. Such a content addressible associative disk appears to be a winner in certain situations (e.g. high percentage of complex transactions). However, it has a drawback that it is not oriented toward the use of a main memory cache.

For small data bases (i.e. data entirely resident in the cache) and data bases with a considerable amount of locality (i.e. data mostly cache resident), MUFFIN should outperform an associative disk. The reason is that RAM is substantially faster than associative secondary storage for many DML commands.

Also, for simple transactions MUFFIN should perform comparably at smaller hardware cost. For example, finding the salary of an employee from a relation keyed on employee name can be done by an associative search of a disk or by an indexed look-up. Depending on the assumptions made about the comparison, either tactic can be made faster. However, the point to be made is that they are in the same performance ballpark.

Lastly, it is not clear how to cheaply make an associative disk resilient to hardware failures.

The point to be noted in summary is that associative processing at the disk level and caching each perform well but in different environments. Both tactics will presumably find application in the areas where they have an advantage.

3.5 Two or More Processors in Tandem

Condider the possibility of a D-cell as noted in Figure 5. Here, some portion of the work is done by processor P1 and some by P2. These could be two levels in a more general hierarchy such as INFOPLEX [MADN79]. There are two conceivable ways to divide data base processing among two such processors.

a) Query deccomposition runs in the upper processor; everything else runs in the lower processor.

This is the approach advocated in [STON78]. However, only perhaps 5 percent of the total work is done by the upper processor; the rest is done in the lower one. Only a hierarchy of one upper processor and several lower processors (perhaps ten) generates a reasonable work distribution. Resiliency issues discussed in Section 3.3 argue against such a hierarchy.

b) The access methods run in the lower processor while all other code resides in the upper processor.

One possibility would be to support the following calls being passed between the two processors:

```
GET_PAGE(key)
GET_NEXT_PAGE
INSERT(record)
DELETE(record_id)
REPLACE(record_id, new_values)
```

Here, P2 would have a one page buffer and would have the software to search that page and do all higher level processing. Note clearly that P2 must buffer one page and not simply one record. The other alternative generates a large number of calls between the two processors when sequentially examining a single page. Such a number of calls increases overhead dramatically.

One the other hand, if P2 has a buffer pool larger than one page, then it must run the entire access methods. This will leave P1 nearly emasculated with almost nothing to do other than those functions normally assocciated with a devive driver in contemporary operating systems.

As a result, it appears that a one page buffer makes the most sense. Under these conditions, Table 1 indicates the approximate work distribution between the two processors.

	P1	P2
simple transactions	10 percent	90 percent
complex transactions	20 percent	80 percent

Work Distribution

Table 1

Although these numbers are speculative, they are based on the following considerations. The expensive action in simple transactions is the fixed overhead associated with start-up and bookkeeping. On the other hand, the expensive action for complex transactions is searching a page. These are both (inevitably) in the upper processor.

As a result there is a poor work distribution and little performance gain with an access method approach. In general, tandem decompositions have both a problem with work distribution and with the fact the tasks cannot be arbitrarily decomposed because of the fixed overhead to communicate between and synchronize multiple processors.

3.6 Two or More Processors in Parallel

Figure 6 indicates what appears to be the desirable architecture for this situation. Although at first glance this resembles RAP [SCHU78] or DIRECT [DEWI78], there are four fundamental differences.

a) One (or maybe two) auxiliary processors is all that can be effectively utilized. RAP and DIRECT suggest use of a much larger number. This point is further discussed in the next section.

b) The auxiliary processors share primary memory with the controlling processor. The RAP and DIRECT proposals suggest having separate memory for each processor.

c) The auxiliary processors should be capable of ONLY accepting a search predicate and a page number in the buffer pool and marking (somehow) the records on that page which satisfy the search criteria. The auxiliary processors in

RAP and DIRECT are able to execute a richer set of commands, which include updates.

d) The controlling processor manages the buffer pool and runs the access methods. Any buffer pool in RAP or DIRECT appears to be in the separate memory of the auxiliary processors.

Consequently, the task of the controlling processor is to run the access methods, fetch pages, manage any locking and recovery which is done, and reformat and dispose of any marked pages which must be rewritten. The auxiliary processor(s) simply searches pages.

For complex transactions such a scheme may win by a factor of two or three (as discussed in the next section). If it clearly appears to be a winner, we may experiment with such a parallel collection of processors. Note clearly that primary memory should not be interleaved location by location in the traditional way but page by page. In this way there will be virtually no contention for main memory.

We conclude this section by commenting that both RAP and DIRECT have separate memories (point b) above) and write changes to that memory (point c) above). Any system with these two features suffers from the following difficulties.

a) Loading and Unloading the separate memories may be a performance bottleneck.

b) Crash recovery may be difficult.

c) A main memory cache for a disk cannot easily be used.

d) Sorting cannot easily be a tactic utilized for query processing; in fact, neither system suggest it. Hence, joins are processed by algorithms with complexity N**2/K where N is the number of pages in each relation being joined and K is the number of processors.

One the other hand, MUFFIN can use sort/merge tactics and achieve complexity of 2N*LOG(N). Sorting will win unless K is very large.

IV MUFFIN PERFORMANCE

In this section we indicate the performance possible in MUFFIN in a few situations. We also indicate ballpark numbers for how fast various components should be. Throughout, we will make a collection of assumptions which are at least vaguely realistic. Prior to building a prototype MUFFIN there is little more that we can do. These assumptions now follow.

a) The overhead for an A-cell to send or receive a message is 5000 CPU instructions.

2) The overhead for a D-cell to send or receive a message is 100 instructions.

3) The overhead per QUEL command in an A-cell for "master INGRES" processing is 100,000 instructions (200,000 if parsing and validity checking is done at run time).

4) The overhead per command for "slave INGRES" processing in a D-cell is 25,000 instructions.

5) The time to search a 4096 byte page for qualifying tuples in a D-cell is 25,000 instructions.

6) The wall clock time in a D-cell to fetch a page from disk randomly is 30 msec.

7) The wall clock time in a D-cell to fetch a page from disk sequentially is 10 msec.

We now indicate the performance of a D-cell under the above assumptions for the following situations.

a) Simple queries -- data in the cache -- few records examined

Here the D-cell must simply pay the fixed overhead for a command (25,000 instructions) and there is no I/O activity regardless of the number of disk drives.

b) Simple queries -- data on the disk -- few records examined

Here, we assume that two random disk reads must be done (60 msec.). This can be thought of as one directory page and one data page under the assumption that the root node of the directory in in the cache. Again, 25,000 D-cell instructions must be executed.

c) Complex queries -- many records examined

Here, many sequential reads will be performed each requiring 10 msec. of wall clock time. Also, the fixed overhead will be inconsequential. What will determine performance is the 25,000 instructions per page to search it.

Table 2 now indicates the performance which can be expected in each situation and what speed CPU in mips (million instructions per second) is required to achieve it.

	number of disk drives		
	1	2	3
simple cache queries	-------40*speed of the CPU in mips -------		
simple disk queries	17 XACTS/sec. (.42 mips)	34 XACTS/sec. (.84 mips)	51 Xacts/sec. (1.26 mips)

complex queries	100 pages/sec. (2.5 mips)	200 pages/sec. (5.0 mips)	300 pages/sec. (7.5 mips)

D-cell Performance

Table 2

Performance of a D-cell appears to be reasonable with a 1-2 mip processor. The only situation where this is too slow is complex queries. Even there, two or three such machines are all that can be effectively utilized. This is the reason for suggesting in Section 3.6 that there will only be a small number of auxiliary processors in parallel.

For the present we will continue the discussion under the assumption that a D-cell employs a PDP-11/70 class machine with a speed of about 1 mip and turn to the bandwidth needed on the bus.

For simple disk transactions, we assume that four messages must be sent on the bus, that an acknowledgement is "piggy-backed" on the next message and that a message is 200 bytes in length.

If so, under the assumption that a D-cell has one disk drive, the bandwidth needed is:

B = (200 bytes/message)*(4 messages/transaction)*
    (17 transactions/sec/D-cell)*

(number of D-cells)

i.e.

B = (13,600 bytes/sec)*(number of D-cells)

Hence, 74 D-cells can be supported on a 1 mbyte/sec. bus. If a processor has more than one disk drive or data is resident in the cache, this number drops. However, it should be clear that many D-cells can be supported on a very ordinary bus (i.e. a few megabits/sec.)

We now turn to complex transactions. The solution to complex transactions involves moving data and executing local transactions. In the algorithms of [EPST78], a D-cell (or any slave INGRES) ALWAYS performs a join or a restriction and a projection prior to moving any data. In such a computation, we assume that a D-cell must process four pages to obtain one which it will ultimately move. Hence, we assume that any result is 1/4 the size of the object from which it was extracted. Hence, a D-cell examines 40 pages per second and generates 10 pages per second for transmission, i.e. 40,960 bytes/sec. Clearly, 25 D-cells will saturate a 1 mbyte/sec. bus. Moreover, because transmission may occur in bursts, saturation may occcur somewhat sooner.

We turn finally to the speed of A-cells. For simple transactions an A-cell can do 5-10 transactions per second times the CPU speed in mips. It appears to be reasonable to assume that the number is 5 for a 1 mip CPU.

We can summarize our ballpark numbers by noting that for simple disk transactions with the property that each transaction involves only one D-cell, a MUFFIN system of 74 D-cells, 252 A-cells and a 1 mbyte/sec. bus can execute 1258 transactions per second. On the other hand, a data base design which has the property that each transaction must involve all D-cells will result in the above MUFFIN system processing only 17 transactions per second. Hence, the success of MUFFIN for simple transactions requires a data base design which puts all data needed by most transactions at a single site.

For complex transactions a 1 mbyte/sec. bus will not saturate until 25 D-cells are involved. Moreover, the algorithms in [EPST78] can attempt to maximize the number of processors which are involved in processing a command. For such transactions it is reasonable that up to 25 times the performance of a single machine system can be obtained.

V COMMUNICATION BETWEEN PODS

We now consider the issue of multiple pods connected by a "long haul" network such as the ARPANET. The argument in favor of running a different protocol on this net than on the local net are coherently summarized in [CLAR78]; hence, one machine in each pod must serve as a gateway and convert between the protocols.

From a software standpoint, distributed INGRES software simply sees more nodes in a distributed data base system. Only the query processing optimization tactics must know that communication between some cells is expensive and others is cheap. Hence, the algorithms of [EPST78] must be expanded to include this knowledge and relevant gateway code must be written. Other than these items, a "long haul" network is transparent.

VI IMPLEMENTATION STATUS

The physical environment in which the MUFFIN prototype will live is indicated in Figure 7.

We have two VAX's which will serve as A-cells and run the UNIX operating system and can dedicate (at least for short periods of time) an 11/70 to serve as a D-cell. We will attempt to "shoehorn" slave INGRES code into 128K bytes and run it in kernal mode. Perhaps we will convert D-cell software to run on a Zilog Z-8000 class CPU sometime in the future.

These three machines are connected by a packet switched local net [ROWE79] based on LNI hardware from University of California, Irvine [CLAR78]. It will run in excess of 1 mbit/sec and support up to 2**16 processors. Hence, modular expansion of a MUFFIN system is straight forward. Lastly, the ARPANET will serve as the long haul network.

The current status of the software is that distributed INGRES runs in this environment and processes transactions against a distributed data base. Some of the optimization tactics in [EPST78] (and some others) have been implemented. Concurrency control and the updating of multiple copies of data have been thought out [STON78], but implementation has not yet begun. It should be noted that resiliency (goal 5 from Section 1) will be provided by supporting multiple copies of objects. This should be contrasted with the "mirrored disk" approach of the TANDEM T-16 system.

Finally, software for slave INGRES on a dedicated D-cell involves substantial changes to the innards of INGRES. These changes are just now commencing.

REFERENCES

[BANE78] Banerjee, Jayanta et. al., "Concepts and capabilities of a Database Computer," TODS, Vol 3, No. 4, December 1978.

[BLAS77] Blasgen, M. and Eswaren, K., "Storage and Access in Relational Data Base Systems," IBM Systems Journal, December, 1977.

[CLAR78] Clark, D. A. et. al., "An Introduction to Local

Area Networks," IEEE Transactions on Communications, November, 1978.

[COPE73] Copeland, G. P. et. al. "The Architecture of CASSM: A Cellular System for Non-numeric Processing," Proc. First Annual Workshop on Computer Architecture, 1973.

[DESP78] Despain, A. M. and Patterson, D. A., "X-Tree: A Tree Structured Multi-processor Computer Architecture," Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[DEWI78] Dewitt, D. J., "DIRECT - A Multiprocessor Organization for Supporting Relational Data Base management Systems," Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[EPST78] Epstein, R. S., et. al., "Distributed Query Processing in a Relational Data Base System," Proc. 1978 ACM-SIGMOD Conference on Management of Data, Austin, Texas, May, 1978.

[FULL77] Fuller, S. H. et. al., "A Collection of Papers on CM*: A Multi-Microprocessor Computer System," Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., February, 1977.

[HAWT79] Hawthorn, P. and Stonebraker, M., "Use of Technological Advances to Enhance Data Base Management

System Performance," Proc. 1979 SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[HELD75] Held, G. D. et. al., "INGRES - A Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975.

[LIN 76] Lin, C. S. et. al., "The Design of a Rotational Associative Memory for a Relational Data Base Management Application," TODS 1, 1, March 1976.

[LORI77] Lorie, R. A. and Wade, B. W., "The Compilation of a Very High Level Data Language," IBM Research, San Jose, Ca., RJ2008, May 1977.

[MADN79] Madnick, S. E., "The INFOPLEX Data Base Computer: Concepts and Directions," COMPCON 79, Sanfrancisco, Ca., February 1979.

[MENA78] Menasce, D. A. et. al., "A Locking Protocol for Resource Coordination in Distributed Systems," Proc. 1978 SIGMOD Conference on Management of Data, Austin, Texas, June 1978.

[RITC75] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System," CACM, June 1975.

[ROTH77] Rothnie, J. B. and Goodman, N., "An Overview of the Preliminary Design for SDD-1: A System for Distributed Data Bases," Proc. 2nd Berkeley

Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., May 1977.

[ROWE79] Rowe. L. A. and Birman, K., "Architecture of a Local Network with Broadcast Facilities," in preparation

[SCHU78] Schuster, S. A., et. al., "RAP. 2 - An Associative Processor For Data Bases," Proc. Fifth Annual Symposium on Computer Architecture, 1978.

[SELI79] Selinger, P. G. et. al., "Access Path Selection in a Relational Data Base System," Proc 1979 SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[SLOT70] Slotnick, D. L., "Logic Per Track Devices," in Advances in Computers, Vol 10, Academic Press, 1970.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

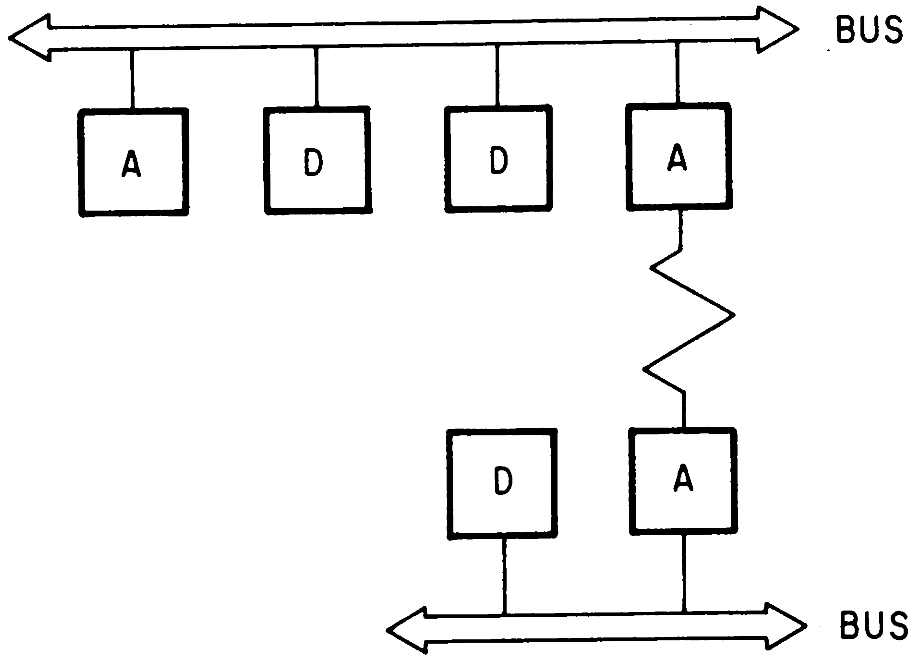
[STON78] Stonebraker, M., "A Distributed Data Base Machine," Electronic Research Laboratory, University of California, Memo No. M78-55, June 1978.

[STON78b] Stonebraker, M., "Concurrency Control, Crash

Recovery and Consistency of Multiple Copies of Data in a Distributed Data Base System," Proc. 3rd Berkeley Workshop on Distributed Data Bases and Computer Networks, San Francisco, Ca., August, 1978.

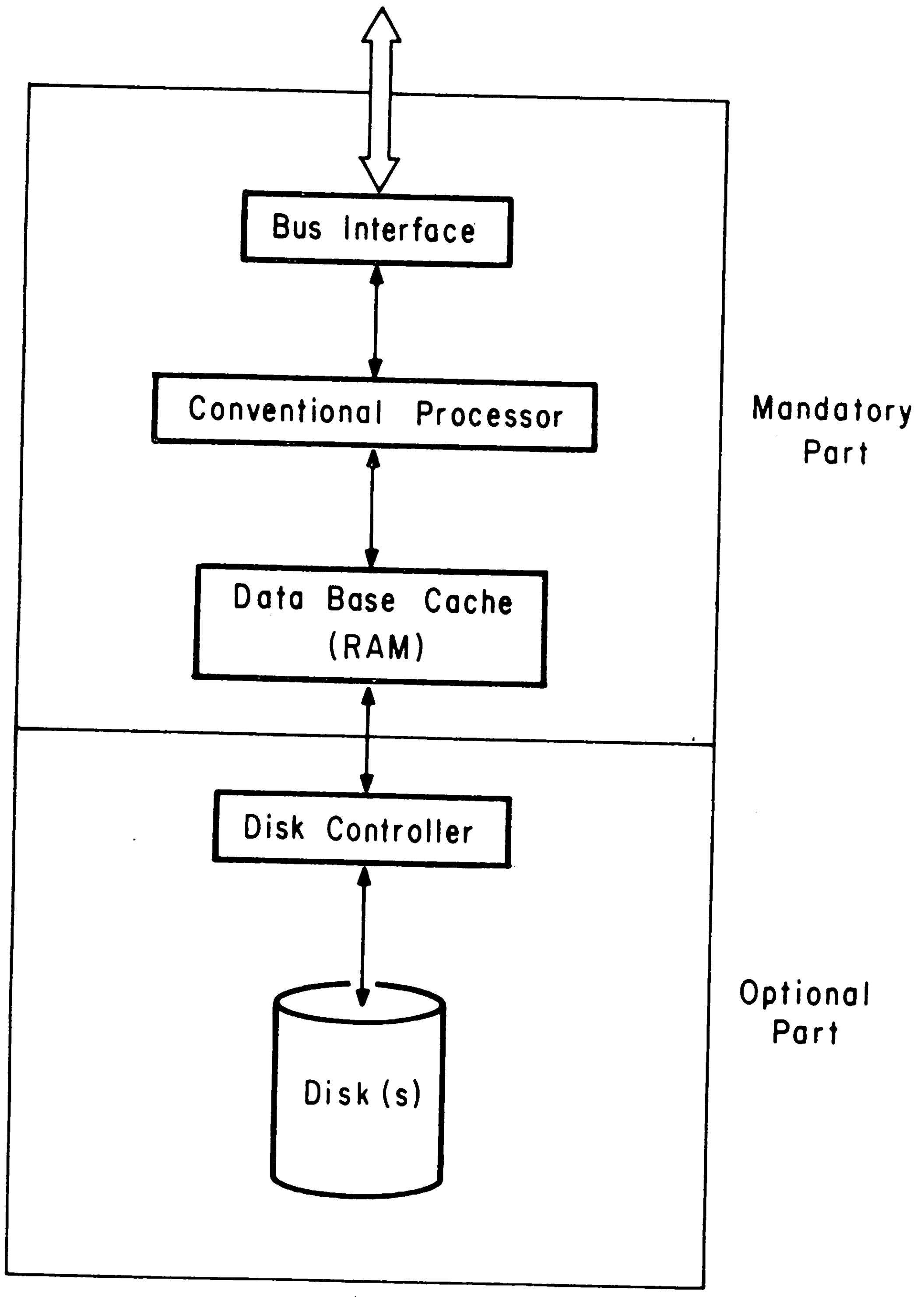
[STON79] Stonebraker, M., "Requiem for a Data Base System," Electronics Research Laboratory, University of California, Memo M79-10, January 1979.

[WONG76] Wong, E. and Youssefi, K., "Decomposition - A Strategy for Query Processing," TODS 2, 3, September 1976.



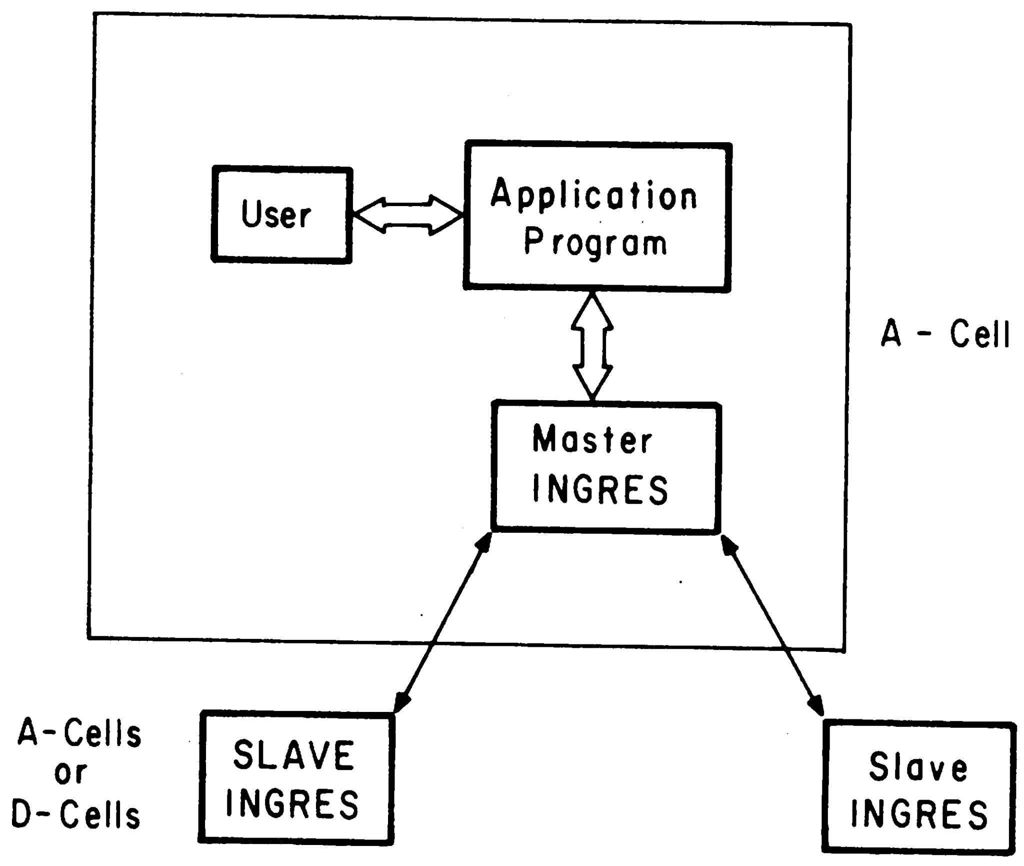
An example of the MUFFIN architecture

Figure 1



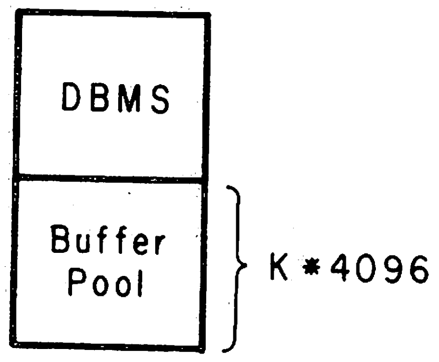
A D-cell

Figure 2



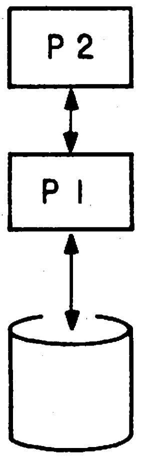
MUFFIN Run-Time Environment

Figure 3



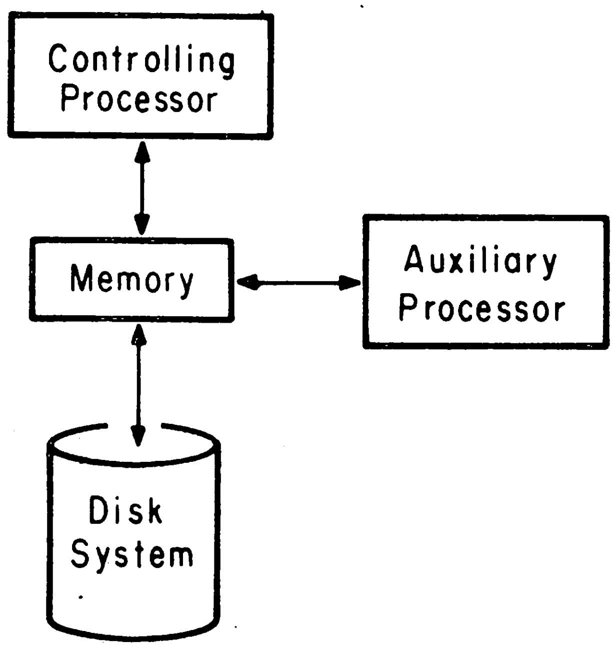
Layout of a D-cell

Figure 4



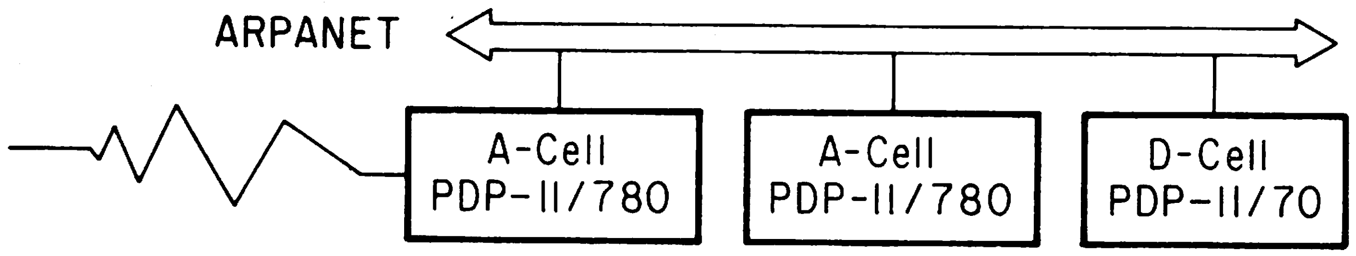
A Tandem Interconnection

Figure 5



A Parallel Architecture

Figure 6



The MUFFIN Hardware

Figure 7