PERFORMANCE ENHANCEMENTS TO A RELATIONAL DATA BASE SYSTEM

by

Michael Stonebraker, John Woodfill, Jeff Ranstrom
Marguerite Murphy, Marc Meyer, and Eric Allman

Memorandum No. UCB/ERL M81/62

27 August 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research Sponsored by the Air Force Office of Scientific Research Grant 78-3596 and the U.S. Army Research Office Grant DAAG29-79-C-0182.

PERFORMANCE ENHANCEMENTS TO A RELATIONAL DATA BASE SYSTEM

by

Michael Stonebraker, John Woodfill, Jeff Rengstrom
Marguerite Murphy, Marc Meyer, and Eric Allman
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

In this paper we examine four tactics commonly advocated to make a data base management system run faster. These are: use of dynamic compilation, use of microcoded routines, use of a special purpose file system and use of a special purpose operating system. All tactics were applied to the INGRES data base management system, and in most cases benchmark timings are reported to indicate the success or failure of the technique.

I INTRODUCTION

Common folklore indicates the existence of a variety of techniques which can be used to make a data base system run faster. These include compiling commands, enhancing the file system, lowering the cost of commonly used system

calls, judicious use of microcode, using special purpose hardware, clustering multiple record types in a single operating system file, etc.

In this paper we report on the application of four of these tactics in the environment of the INGRES relational data base system [STON76, STON80]. To the extent that INGRES is representative of other systems, the results may have general applicability. In the next subsections we outline the tactics we studied.

1.1 Compilation

INGRES is currently an interpretive data base system. Consequently, commands are parsed, validated and then an access plan is built and executed, all at run time. Although ad-hoc commands from a user at a terminal have only the notion of run time, commands from a host language program can have some of the above functions performed at compile time. As a result, run-time overhead would be reduced. For example, System R [ASTR76, BLAS79] compiles commands into machine language programs to be executed at run time. The various options have been sketched in [KATZ78] and include several possibilities between the INGRES pure interpreter and the System R pure compiler.

Performance studies in [HAWT79] suggest that parsing and validating an INGRES command at compile time would be a major win for so-called overhead intensive queries (i.e. short commands). In fact, the parse-at-run-time strategy is

one of the major mistakes in the design of INGRES noted in [STON80]. Implementors of future relational systems should avoid this mistake, and it is possible that INGRES will be modified to parse at compile time at some point in the future.

However, in longer transactions parsing at compile time would have little impact. The overhead of parsing and validating a command is dominated by the effort needed for data searching. On long commands about 95 percent of the CPU time is spent by INGRES inside an inner loop [HAWT79] consisting of a large case statement which compares a record to a template and reports whether the record satisfies the qualification indicated by the template.

To reduce the overhead of record interpretation we could incorporate a pure compiler in INGRES, but this would involve a complete rewrite of the system and is infeasible. On the other hand, we feel that dynamic compilation is a viable alternate strategy. Hence, we would compile a procedure to perform record selection while processing an individual command. As a result, instead of paying N interpretations for N records, we pay the cost of 1 compilation followed by N executions of the compiled code.

The use of dynamic compilation has several advantages compared to generating machine code at compile time. First, access path selection can continue to be done at run time. The code to accomplish this is very complex, and an inter-

preter for it is much easier to build than a compiler. In addition, access path selection at run time can make use of current statistics concerning relation sizes and the actual sizes of partial results when planning the next step to take [EPST80].

In Section II of this paper we report on the design of our dynamic compiler and give benchmark results concerning its performance.

1.2 Microcode

The INGRES project possesses a VAX 11/780 computer with a user writable control store. Hence, we could write microcode for commonly executed routines. The folklore claims that a major speed improvement can result from judicious use of microcode. At least two sources contribute to this potential speed-up. First, one can replace multiple machine language instructions by a single one and save the instruction fetch portion of each execution cycle. Perhaps more importantly, one can code a better algorithm in microcode than is possible with the machine instructions provided by the computer designer. This area has been fully exploited in speeding up context switches and subroutine calls in several environments.

Of course, the problem with microcode is that few tools are available to assist in its construction. For example, we were unable to locate a microcode assembler which would run under UNIX, and the documentation on how to use the

microstore is nearly impenetrable. Because of the poor tools and documentation, we were unsuccessful in producing a microcode version of INGRES which could be benchmarked. However, in Section III we report on our experiments with profiling INGRES to isolate high traffic routines. Then, we estimate the size and expected speed-up of INGRES with a microcode assist.

1.3 File Systems

It is often asserted that the operating system is the real villan which slows down the performance of a data base system [STON80a]. What is usually meant is that system calls and task switches are too slow or that the file system is not designed to adequately support the needs of a data base system.

In the UNIX environment the file system was designed to serve time sharing users [RITC75, BSTJ78]. Hence, it is easy and efficient to create and destroy files and to change their length dynamically. However, UNIX randomly allocates disk blocks to a file. As a result logically adjacent blocks in a file are not necessarily physically close. Since INGRES does a substantial amount of sequential access [HAWT79], random allocation is undesirable. Moreover, UNIX uses a collection of indirect blocks to map logical pages to physical blocks. These indirect blocks must be read during an access and increase the overhead associated with reading or writing a file. Lastly, the block size used by UNIX

(1024 bytes) appears rather small for data base use.

Consequently, we have designed our own relation storage system that operates on top of a "raw" disk. This software efficiently supports the storage of relations and allows reads and writes to them without going through the UNIX file system. In Section IV we briefly discuss the design of this software and then show via benchmark studies the impact which it has on the performance of the overall system.

1.4 General Purpose Operating System

Operating systems usually support processes with a lot of state information. As a result switching tasks is often an expensive operation. Moreover, on most current systems interprocess messages are very expensive. Since a data base system must use both facilities extensively or implement its own multitasking system in user space [STON80a], the operating system may stand in the way of efficient data base operation.

To study this issue we designed a small special purpose operating system, MANGOS. This software runs on a bare machine and supports only the facilities that a data base system and a network manager need. It has no mechanisms to swap processes out to the disk since both software systems want to be permanently resident in main memory. Processes are very lightweight objects, and messages are simply pointers to buffers that can be exchanged with only a few instructions of overhead.

In Section V we indicate the design of MANGOS and make estimates concerning its performance. Since it is not yet completely coded, benchmark studies have not been run in this environment.

1.5 The Benchmark

In order to test our dynamic compiler and raw device file system we constructed the following benchmark. It consisted of two sections, the first was designed to test commands which would sequentially scan an entire relation which was stored as a heap while the second tried to illustrate performance when a keyed access path could be used.

The first section consisted of four queries using the following PARTS relation:

PART(pnum, pname, pcolor, pweight, qoh)

It contained 34 byte tuples and was stored as a heap. Runs were made with 1, 210 and 3360 tuples occupying respectively 1, 8 and 118 pages.

(a)
```
retrieve (a = 1)
```

(b)
```
range of p is PARTS
retrieve (p.pnum)
```

(c)
```
retrieve (p.pname, p.pnum)
where p.pnum > 3 or p.pname = "processor"
  or p.color = "black"
```

(d)
```
retrieve (p.pnum) where p.qoh < log(9.5) * p.weight
```

Query (a) is intended to show the cost of compilation for a very short query. In this case, the query require no database accesses and its interpretation is extremely cheap. On the other hand, compiling such a query should only generate additional overhead.

Queries (b) through (d) require a complete sequential scan of the PARTS relation. Moreover, c) and d) have a complex qualification which should be expensive to interpret. A compiler should perform well in these situations.

The second porion of the benchmark deals with a PARTS relation which has tuples widened to 200 bytes with a filler field. Moreover, PARTS contains 3360 tuples and was stored indexed sequential (isam) with pnum as a key. The data level of this structure contained 646 1K pages each with 5 tuples and 130 1K overflow pages each with a single record. This situation models a perfectly built keyed structure for 3230 tuples followed by 130 random inserts.

(e)
```
retrieve (p.qoh) where p.pnum < 2500
```

(f)
```
range of q is PARTS
retrieve (p.qoh, q.qoh) where p.pnum = q.pnum
```

(g)
```
retrieve (p.qoh, q.qoh) where
p.pnum = q.pnum and p.color = "black"
```

(h)
```
retrieve (p.qoh) where
p.pnum = q.pnum and p.pnum <2500
```

(i)
```
retrieve (p.qoh, q.qoh) where p.pnum = q.pnum
and p.color = "black" and p.qoh < log(9.5) * p.weight
and q.qoh < log(9.5) * p.weight
```

Query e) illustrates a range search on a portion of PARTS. Then commands f) - i) suggest various natural joins all of which connect PARTS to itself using the keyed field. These queries are intended to be illustrative of complex commands and differ in the amount of added qualification. In all cases performance numbers were generated for both an isam structure and for a PARTS relation hashed on pnum. Since the results are similar, we only present the isam numbers.

All numbers will be in seconds and come from a DEC PDP-11/780 (VAX) computer running the UNIX operating system.

II A DYNAMIC COMPILER

In this section we sketch the design of our dynamic compiler and then give results on the performance improvement obtained.

2.1 Design of DC-INGRES

Dynamically Compiled INGRES (DC-INGRES) is entirely concerned with compiling commands that affect a single relation. The code to decompose multi-relation commands into a sequence of single relation commands is unaffected.

For example, the following command finds all employees under 30 or in the toy department or who make less than $20,000.

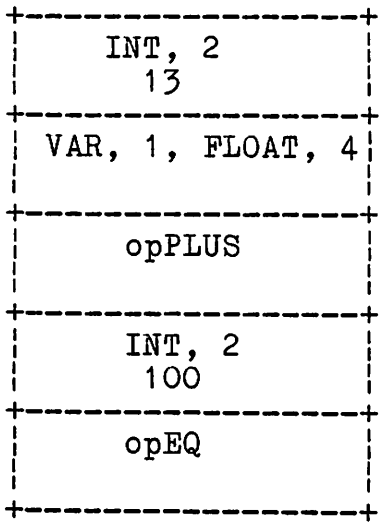
```
RANGE OF E IS EMPLOYEE
RETRIEVE (E.name) WHERE E.age < 30  OR
                        E.dept = "toy"  OR
                        E.salary < 20000
```

Current INGRES algorithms will scan a subset of the employee relation record by record, interpreting the above qualification for each one to establish whether the record qualifies.

The implementation is via a straightforward stack-oriented interpretation of a postordered list of symbols representing the command. Figure 1 gives an example of the data structure used for the qualification "WHERE 13 + E.salary = 100". The second entry in the stack indicates that salary is the first field in EMPLOYEE and is a four byte floating point number. The other entries are self-explanatory.

This is a high overhead approach to record evaluation. For example, if there are 1000 tuples that must be checked against this qualification, then the code for EACH of the operators will check its arguments each of the 1000 times to determine the type of its operands. That is to say, the code for opPLUS will determine that it must add an integer to a float anew for each tuple. Moreover, the integer 100 will be converted to a float 1000 times so that opEQ can compare two operands of the same type.

To alleviate this overhead, the DC-INGRES compiler produces a form of code loosely based on direct threaded code [BELL73, DEWA75]. The compiler takes as input a list such

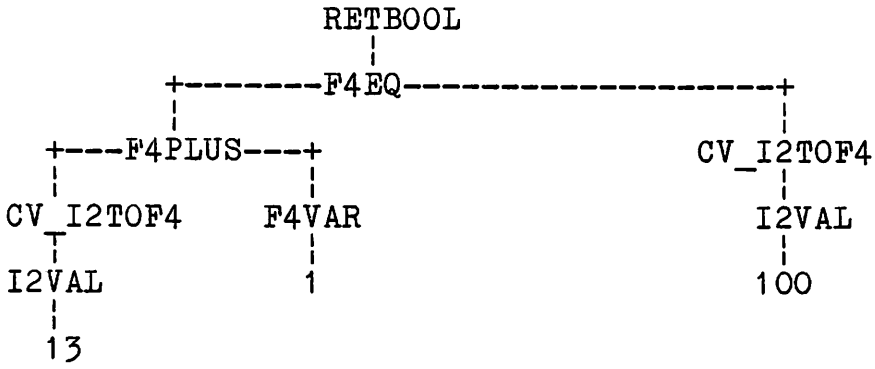


The Qualification "where 13 + EMP.salary = 100"

Figure 1

as the one in Figure 1, then builds a tree-formatted intermediate code data structure with it. In this tree, shown for the above qualification in Figure 2, the nodes are specialized operators, and their descendants are their operands. Hence, all necessary conversions are built into the tree and no decision making concerning types is done during tree evaluation. Moreover, the code required to do necessary conversions is pointed to directly by a node. For example, the node F4PLUS is a pointer to code to add two four byte floats and return a four byte float.

Two improvements to this structure and the evaluation machine which processes it have been implemented. First, the evaluation machine has a collection of registers capable of holding a value of any type. Each operation is given as



The Intermediate Code Tree

Figure 2

arguments the registers of its operands and where to put its result. Second, DC-INGRES processes constant expressions as it builds the tree, generating code to calculate the expression and to leave the result in a register, then calls for the execution of the code, leaving the constant values in registers it has allocated for the purpose. Consequently, the right-most branch of the tree in Figure 2, converting 100 to a float, is done once during construction of the tree.

Although DC-INGRES does not produce in-line machine code, we feel that the performance penalty for our threaded code solution may not be large. The additional overhead is perhaps two or three instructions per operator in the tree. Since most nodes require a considerable number of instructions to implement, the threaded code penalty may be as little as 5-10 percent. On the other hand, our compiler was dramatically easier to construct than one which produced machine code directly.

2.2 The Performance Results

This section discusses the results obtained running the benchmark indicated in Section 1.5 for standard INGRES and DC-INGRES. Two statistics were measured, elapsed wall clock time with the benchmark as the only active task and CPU time spent inside INGRES.

query	relation size	DC-INGRES wall clock	INGRES wall clock	IMPROVEMENT
a	0	.9	.9	0%
b	1	.9	.9	0%
b	210	1.2	1.3	8.3%
b	3360	5.0	5.7	14.0%
c	1	1.2	1.2	0%
c	210	1.7	2.0	17.6%
c	3360	7.6	11.5	51.3%
d	1	1.0	1.0	0%
d	210	1.5	1.8	20.0%
d	3360	5.5	11.5	109.1%
e	3360	17.1	18.7	9.4%
f	3360	52.8	63.5	20.3%
g	3360	31.5	33.3	5.7%
h	3360	47.5	54.5	14.7%
i	3360	32.3	34.4	6.5%

Wall Clock Time

Table 1

query	relation size	DC-INGRES CPU time	INGRES CPU time	IMPROVEMENT
a	0	.087	.087	0%
b	1	.123	.112	- 9.5%
b	210	.353	.401	13.7%
b	3360	3.88	4.73	21.8%
c	1	.208	.195	- 6.4%
c	210	.605	.823	36.1%
c	3360	6.37	10.3	61.9%
d	1	.177	.162	- 8.5%
d	210	.433	.812	87.3%
d	3360	4.25	10.6	148.3%
e	3360	6.43	8.96	39.4%
f	3360	42.4	53.1	25.2%
g	3360	10.6	13.1	23.2%
h	3360	33.1	41.9	26.3%
i	3360	11.1	14.1	26.7%

INGRES CPU Time

Table 2

2.4 Conclusions

The results show that for commands affecting a single tuple the compiler costs somewhat more CPU time (about .012 seconds) but generates the same response time. In general, DC-INGRES cuts the cost of processing a record substantially (from .0031 to .0019 sec. for query c) with 3360 tuples). Consequently, if one evaluates more than about ten records, one would expect DC-INGRES to outperform regular INGRES. The win is very dramatic for complex one relation commands, for example d) and e), when a large number of records are examined. In general, the improvement in response time is less impressive because INGRES is not CPU bound in all cases.

A multi-relation command is decomposed by INGRES into a sequence of one relation commands. In f) through i) they are generally of the form

WHERE pnum = constant

and only a few records are evaluated for each one since PARTS is keyed on pnum. However, DC-INGRES notes that a sequnece of identical commands are generated differing only in the constant used. Consequently, it reuses the compiled form of the query and substitutes new constants. Hence, the .012 sec. does not need to be paid each time. As a result an improvement is still possible in all multi-relation commands in the benchmark.

III MICRO-INGRES

In order to evaluate the feasibility of a microcode version of INGRES (M-INGRES) we first profiled the INGRES run-time code to find high traffic routines as candidates for conversion to microcode. Then, we hoped to recode some of them in microcode so we could benchmark M-INGRES to test its speed. In the section we report on our findings.

3.1 Profiling

In order to identify the high traffic routines, we ran the benchmark with the profiler enabled. This package indicates the percentage of CPU time inside each procedure of the INGRES run time code. The major portion of INGRES time

is spent inside the code that evaluates so-called one variable commands (OVQP). A substantial fraction of this code is the record selection code discussed in Section 2. Table 3 indicates the routines with more than 3.0% of the run time CPU cycles.

%time	routine	short explanation
17.1	GETSYMBOL	get symbol from list
14.8	INTERPRET	interpret list
4.4	LOG	logarithm
4.3	READ	read system call
3.9	BMOVE	block memory move
3.7	GET	get tuple
3.1	FWRITE	write to file
3.0	SCAN	scan a relation

High Traffic Routines

Table 3

A brief description of each of these routines now follows.

GETSYMBOL: This routine fetches symbols from the data structure representing the qualification indicated in Figure 1. The symbols themselves are triples consisting of a type, a length, and a value. The representation of the value depends on both the type and length. Within the subroutine, the legality of the type and length are checked and an indication of whether the list has been exhausted is returned to the calling routine. This routine is 105 lines of C.

INTERPRET: This is the main routine of OVQP. It processes tuples against a qualification structured as in Figure 1 and

uses GETSYMBOL to fetch the symbols. The code is structured as a loop which first removes a symbol from the list, then processes it using a series of nested switch statements. This routine is 420 lines of C.

LOG: This is the logarithm function from the C procedure library. It is present because the benchmark contains commands with a logarithm in them.

READ: This is the C procedure which formats a user disk read and does the operating system call. This time does not include the time spent inside the kernal executing the call.

BMOVE: This routine moves a block of data from one place in main memory to another. It uses the VAX block move instruction.

GET: This is the access method call to fetch a tuple. It is called by the routine SCAN when another tuple is required for evaluation.

WRITE: This is the C procedure which disposes of output. The benchmark contains retrieve commands, the output of which is directed to a file using this routine.

SCAN: This routine sets up a scan of a relation as a result of access path selection code. It calls GET to fetch a tuple and then calls INTERPRET to discover if the tuple qualifies.

3.2 Design of M-INGRES

The approach used to microcode the VAX was to view the user microstore as a (fast) C subroutine and use the standard C subroutine linkage as the interface. This has the advantage of allowing one to pass arbitrary parameters to the microcode in a straightforward, well-defined way. It also allows us to call the microcode directly from C instead of having to patch in assembler code. The overhead to enter and exit the user microstore (ie the overhead exclusive of the C subroutine entrance/exit) was timed at 3.3 microseconds.

3.3 Performance of M-INGRES

After laborious effort we were able to get an integer add instruction to work correctly from the user microstore. Most of the problems we encountered concerned inadequate documentation and support tools. However, we gained enough experience to estimate values for the columns in Table 4.

In that table we first give an estimate for the number of microwords that we would have to write for the two highest traffic routines. This number was obtained by multiplying the number of assembly language instructions in the routine by 16. This is the average number of microwords per instruction in the system microstore which implements the VAX 11/780 instruction set.

The remainder of the table deals with a performance estimate for the resulting microcoded routine. The time savings due to a microcode version were assumed to result entirely from avoiding an instruction fetch on failed branch statements. The one-instruction prefetch mechanism used by the VAX 11/780 was assumed to operate reliably in all other cases. An estimate of the number of failed branch statements was taken to be one half the static count of branch statements in the assembly level code. Since all six routines rarely loop, this number is reasonable. The time for instruction fetch was assumed to be 290ns.; which is the average access time to the VAX 11/780 hardware cache. Hence, the second entry in the table is our estimate of this resulting time savings. Then, we report the results of timing each of the routines by giving the average number of CPU msec. per call. The final two columns respectively give our estimate for the percentage speed up of each routine due to microcode and its overall impact on INGRES performance.

3.4 Conclusions

subroutine	estimated number of microwords	estimated speed-up of microcoded routine [ms]
GETSYMBOL	135 * 16 = 2160	(43/2) * 290 = .006
INTERPRET	514 * 16 = 8224	(119/2) * 290 = .017

Performance of M-INGRES

Table 4

	msec. per call	percentage speed up of routine	increase in overall INGRES performance
GETSYMBOL	0.07	8.6%	1.6%
INTERPRET	0.36	4.7%	0.7%

Performance of M-INGRES

Table 4 (continued)

The VAX instruction set has been designed to encompass all of the common high level language constructs. For example, the C switch statement maps directly into the VAX assembler case statement. Of the high traffic routines studied, none exhibited constructs not already present in the existing assembly language instruction set. It is our assessment that we would not be able to substantially outperform their implementations. Hence, we would not expect to be able to make algorithm improvements which would substantially impact Table 4.

As can be seen, we would gain a total of 2.3% improvement from about about 10K words of microcode. This increase certainly does not justify the time and expense of microcoded enhancements. If the improvement were doubled or even tripled, we would still come to the same conclusion. In addition, the overhead to call a microcoded routine (3.3 microseconds) was not included in the calculations. If it had, a net negative time savings would have resulted.

IV FS-INGRES

This section reports on the design of INGRES enhanced with a special purpose file system. This composite is FS-INGRES which is now described.

4.1 The Design of FS-INGRES

The basic objectives of FS-INGRES are:

(1) To provide an extent based file system, with a minimum extent size of one track. In this way random allocation of disk blocks will be avoided.

(2) To avoid the UNIX requirement of copying data from system buffers to the INGRES cache. This is accomplished by a direct read to user space.

(3) To organize the data on the disk so that high traffic relations, for example the system catalogs, are advantageously placed near the center of the disk surface.

To accomplish these objectives, FS-INGRES implements the notion of a relation on top of a raw disk. Instead of directories and files, there are only data bases and relations. Each relation is stored in up to 8 extents, each of which is a variable size physically contiguous collection of disk blocks. Such extents are allocated as needed, and the minimum extent size is one track. Instead of having a file control block, FS-INGRES stores information on the extents of a given relation in the RELATION relation. This relation contains one row of INGRES specific information for each relation in a data base.

Free space is managed using a bit map which resides at a known location on the disk. Moreover, start-up information, such as the address of the RELATION relation, is also at hard-wired addresses.

Lastly, UNIX supports only synchronous I/O; i.e. a process which issues a disk read does not get control again until the requested data is in main memory. Therefore, it is impossible for FS-INGRES to implement any prefetching policy because it would require asynchronous I/O. As noted in [STON80a] INGRES usually knows which block it will access next and an intelligent prefetch could be realized, but only by modifying UNIX.

As an interim approach which would not require operating system changes, FS-INGRES reads an entire track of data at a time in those case where it is doing sequential processing. In this way it obtains 16 logically contiguous 1024 byte blocks as a result of each read. It is likely that UNIX will be modified to support asynchronous I/O in the future so that a more sophisticated strategy can be implemented.

4.2 Benchmark Results

Tables 5 and 6 give benchmark results for standard INGRES and FS-INGRES and indicate wall clock time and total CPU time with the benchmark as the only executing task.

query	size	FS-INGRES	INGRES	IMPROVEMENT
a	0	0.7	0.9	28.6%
b	1	0.8	0.9	12.5%
b	210	1.1	1.3	18.2%
b	3360	5.2	5.7	9.6%
c	1	1.1	1.2	9.1%
c	210	1.9	2.0	5.3%
c	3360	11.2	11.5	2.7%
d	1	1.0	1.0	0%
d	210	1.7	1.8	5.9%
d	3360	11.0	11.5	4.5%
e	3360	10.8	18.7	73.1%
f	3360	54.2	63.5	17.2%
g	3360	18.0	33.0	85.0%
h	3360	46.0	54.5	18.5%
i	3360	19.4	34.4	77.3%

Performance Comparisons for Wall Clock Time

Table 5

query	size	FS-INGRES	INGRES	IMPROVEMENT
a	0	.083	.087	4.0%
b	1	.122	.112	- 8.2%
b	210	.378	.401	6.2%
b	3360	4.24	4.73	11.6%
c	1	.187	.195	1.0%
c	210	.787	.823	4.7%
c	3360	9.77	10.3	5.5%
d	1	.160	.162	1.0%
d	210	.767	.812	5.9%
d	3360	9.87	10.6	7.0%
e	3360	6.64	8.96	35.0%
f	3360	48.7	53.1	9.2%
g	3360	9.11	13.1	43.8%
h	3360	37.6	41.9	11.4%
i	3360	10.0	14.1	40.3%

Performance Comparisons for INGRES CPU Time

Table 6

4.3 Conclusions

Several conclusions are evident from the above tables. In simple commands FS-INGRES gives better response time than standard INGRES (typically 10-20%) with comparable CPU time. The explanation is probably faster access to the system catalog relations due to track-at-a-time reads. On longer commands which involve sequential access (b, c and d with 3360 tuples) FS-INGRES saves about 10% in CPU time with a lesser response time gain. In these situations UNIX notes sequential access and prefetches the next logical page in advance. Hence, processing the previous page is overlapped with fetching the next one. Consequently, it stays nearly even with FS-INGRES which fetches whole tracks.

Case e) shows dramatic improvement. In this situation INGRES accesses primary pages intersperced with overflow pages. UNIX will note a run of two primary pages and fetch the third one in advance. However, a switch to an overflow page will destroy sequentiality and UNIX will fail to prefetch both the overflow page and the next two primary pages. Consequently, CPU and I/O activity are not consistently overlapped, and standard INGRES suffers poor response time.

On the other hand, FS-INGRES will read a track at a time and capture a whole run of primary pages. It must wait for overflow pages like normal INGRES but not for the next primary page. As a result, there is a dramatic gain in response time.

In command g) INGRES decomposition will first find all black parts as follows:

```
retrieve into temp(p.pnum) where p.color = "black"
```

This command will involve a complete sequential scan of PARTS and is accomplished by reading primary pages interspersed with overflow pages. Moreover, since there are few black parts, processing the rest of the query is quite simple. Consequently, the argument for case e) applies to this query and a dramatic win is observed. Case i) is a similar command and exhibits the same behavior.

Queries f) and h) are dominated by the time to do the actual join. Although PARTS is read sequentially as above and an improvement recorded, this speedup must be spread over considerably more CPU time. Consequently, the percentage improvement is less impressive.

In general, FS-INGRES saves 5 to 40% in CPU time and produces a command-dependent improvement in response time. Future experimentation is needed to determine the impact of parameters not considered. These include the effect of asynchronous I/O (which should benefit FS-INGRES which could then implement an accurate prefetching mechanism), modifying UNIX to store pages contiguously and read a track at a time (which might allow standard INGRES to realize most of the gain in query e), and running the benchmark in a multiuser environment (which would add the effects of disk contention

and multiprogramming to both systems).

V MANGOS

To assess the performance improvement of a special purpose operating system, we designed MANGOS (MArgie's Non General Operating System). In this section we report on its design and estimate its performance. MANGOS is intended to operate on a dedicated processor which receives INGRES commands from another machine.

The main modules in MANGOS are:

(1) the buffer manager which is responsible for handling the buffer pool

(2) the process manager which co-ordinates processes

(3) the device drivers, which handle all of the actual I/O initiation and interrupt handling

(4) INGRES, which executes commands against the local portion of the data base.

(5) The network manager, which receives INGRES commands from the outside world and returns the results of such commands

5.1 Buffer Management

A buffer consists of a 4096 byte page-aligned section of shared user address space and an associated header in a separate data structure. Every header appears on one of the following lists: free, owned by the disk, owned by the

network, disk I/O in progress and network I/O in progress.

5.2 Process Control

The only processes supported by MANGOS are instantiations of INGRES. Such processes execute in a single address space with one copy of the INGRES code and a shared buffer pool. Each process contains a private data area which hold the data structure corresponding to the current command and its state of execution. The number of processes is established at system initialization time and each has an associated process control block (PCB).

Processes are coordinated by moving their PCB's on and off various queues. Each process is initially on an idle queue, then it is moved to the ready queue when allocated to an arriving INGRES command. Thereafter, it cycles between the ready and blocked queue as it does I/O.

An INGRES process invokes the services of the buffer manager via a subroutine call which returns when the action is complete (except in the case of asynchronous I/O where control is returned after I/O is initiated, but not necessarily complete).

Context switches only occur when a process calls the buffer manager requesting service and is placed on the blocked queue. Since processes all execute in the same address space, we need only save the current processor state, find the processor state of the next ready process

and restore it. We have estimated that 10 machine instructions are required to accomplish this task.

5.3 Device Drivers

The disk driver is responsible for all dealings with the disk controller. As such it moves blocks between the disk and the buffer pool in a standard way. The other driver present in MANGOS is the network manager which will be a modified version of COCANET [ROWE79]. COCANET moves blocks between the buffer pool and the network.

5.4 Messages

The only messages in MANGOS are communications among INGRES, the network manager and the buffer manager. Messages are implemented by removing a buffer from one list and placing it on another. In essence, a message is a pointer to a 4096 byte block of storage. The overhead to send or receive a message has been estimated at 9 instructions.

5.5 Performance Estimates

MANGOS will enhance INGRES performance in two ways, namely it will speed up system calls (notably messages and task switches) and support parallelism due to a multiprocessor configuration. We will estimate the impact which would result from faster task switches and messages. It is difficult to obtain statistics for the number of times INGRES blocks (which is the event which would cause a task switch).

Hence, we will use the fact that an approximation for this quantity is the number of disk accesses which UNIX makes on behalf of INGRES. This number is less than the number of pages touched by INGRES because of the fact that UNIX caches disk blocks in main memory and does not need to fetch accessed blocks from the disk which are already in the cache. Table 8 summarizes this information for each command in the benchmark along with the number of messages.

Currently, UNIX/VM on a PDP-11/780 requires 0.270 msec. to perform a task switch and 1.4 msec. to send a message to another process [JOY_80]. (Extensive tuning has made these numbers nearly a factor of 3 smaller than previous UNIX/VM timings.) These would drop to about 10 and 9 microseconds respectively in MANGOS. Consequently, Table 9 contains our estimates for MANGOS performance based on these numbers.

query	size	number of actual reads	number of messages
a	0	0	4
b,c,d	1	5	4
b,c,d	210	13	4
b,c,d	3360	125	4
e	3360	609	4
f	3360	1572	4
g	3360	1210	4
h	3360	1317	4
i	3360	1224	4

Access Statistics for MANGOS

Table 8

query	size	MANGOS CPU time	INGRES CPU time	IMPROVEMENT
a	0	.081	.087	6.9%
b	1	.105	.112	6.3%
b	210	.392	.401	2.3%
b	3360	4.69	4.73	0.8%
c	1	.188	.195	3.6%
c	210	.814	.823	1.1%
c	3360	10.26	10.3	0.4%
d	1	.155	.162	4.3%
d	210	.803	.812	1.1%
d	3360	10.56	10.6	0.4%
e	3360	8.79	8.96	1.9%
f	3360	52.7	53.1	0.8%
g	3360	12.8	13.1	2.3%
h	3360	41.5	41.9	1.0%
i	3360	13.8	14.1	2.1%

MANGOS Performance

Table 9

VI CONCLUSIONS

Loosely speaking the results of this paper can be summarized as follows.

tactic	effect	estimated difficulty
DC-INGRES	no effect for short commands; 25-100% for longer commands	4 man-months
M-INGRES	2.3-6.9%	more than 6 man-months
FS-INGRES	5-40% CPU time 5-50% response time	2 man-months
MANGOS	1-4%	6-12 man-months

Clearly compilation and file system tactics have a high pay-

off and are relatively easy to accomplish. Microcode and a special operating system do provide a performance improvement but not a large one and at very high cost.

It is hoped that these studies will give data base system designers some insight into where to put their own human resources to obtain maximun cost effectiveness.

REFERENCES

[ASTR76] Astrahan, M. M. et. al., "System R: A Relational Approach to Database Management," TODS 2, 2, June 1976.

[BELL73] Bell, James R., "Threaded Code," CACM Vol 16, No. 6, June 1973.

[BLAS79] Blasgen, M., et. al., "System R: An Architectural Update," IBM Research, San Jose Ca., Report RJ2654, June 1979.

[BSTJ78] "The UNIX Time-Sharing System," Bell Systems Technical Journal, Vol 57, No. 6, July 1978 (special issue).

[DEWA75] Dewar, Robert, "Indirect Threaded Code," CACM Vol 18, No. 6, June 1975.

[EPST80] Epstein, R., and Stonebraker, M., "Analysis of Distributed Data Base Processing Strategies," Proc. Sixth Very Large Data Base Conference, Montreal, Canada, October 1980.

[HAWT79] Hawthorn, P. and Stonebraker, M., "Use of Technological Advances to Enhance Data Base Management System Performance," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1979.

[KATZ78] Katz, R., "Compilation in Data Base Systems," Proc. 1978 National Computer Conference, New York, New York, June 1978.

[JOY_80] Joy, W., "Comments on the Performance of UNIX on the VAX," (unpublished working paper).

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

[ROWE79] Rowe, L. and Birman, K., "The Design of COCANET," Proc. 4th Berkeley Workshop on Distributed Data Bases and Computer Networks, Berkeley, Ca., October 1979.

[STON76] Stonebraker, M. et. al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M., "Retrospection on a Data Base System," TODS, September 1980.

[STON80a] Stonebraker, M., "Operating System Support for Data Base Management Functions," CACM, July 1981.