



HARDWARE SUPPORT FOR ADVANCED
DATA MANAGEMENT SYSTEMS

by

Philip M. Neches

5082:TR:83

HARDWARE SUPPORT FOR ADVANCED
DATA MANAGEMENT SYSTEMS

Thesis by
Philip M. Neches

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1983
(Submitted 4 May 1983)

Copyright 1983
Philip M. Neches
All Rights Reserved

ACKNOWLEDGEMENTS

This work owes its existence to the advice and support of many people. David W. Morrisroe encouraged me to begin the work at a propitious time. Professor Ivan E. Sutherland created the department which provided the stimulating intellectual environment in which the work was done.

The Computer Science Department provided financial support through a variety of teaching assignments. My parents, Bernard and Vivian Neches, also provided timely financial support which permitted the work to be finished.

This work resulted from the intellectual and personal guidance of two great thinkers and teachers: Professor Carver A. Mead and Professor Frederick B. Thompson. In particular, I must acknowledge my debt to Professor Thompson who in his years as my advisor instilled in me the courage to encounter and overcome the challenge of research and discovery.

Finally, I must acknowledge the encouragement, support, and devotion of my wife, Teresa T. Young, without which this work would not have been started, much less completed.

ABSTRACT

This thesis considers the problem of the optimal hardware architecture for advanced data management systems, of which the REL system can be considered a prototype. Exploration of the space of architectures requires a new technique which applies widely varying work loads, performance constraints, and heuristic configuration rules with an analytic queueing network model to develop cost functions which cover a representative range of organizational requirements. The model computes cost functions, which are the ultimate basis for comparison of architectures, from a technology forecast. The discussion shows the application of the modeling technique to thirty trial architectures which reflect the major classifications of data base machine architectures and memory technologies. The results suggest practical design considerations for advanced data management systems.

CONTENTS

<u>Chapter</u>		<u>Page</u>
CHAPTER 1	STATEMENT OF THE PROBLEM	1
1.1	BACKGROUND	1
1.2	THIS INVESTIGATION	11
CHAPTER 2	THE MODEL	16
2.1	INTRODUCTION	16
2.1.1	Overview	16
2.1.2	Modeling Technique	18
2.1.3	Implementation	23
2.2	THE CONFIGURATION MODEL	26
2.2.1	The Hardware View	28
2.2.2	System Analyst View	37
2.2.3	User Organization View	39
2.2.4	Detail Output	40
2.3	ARCHITECTURE STUDIES	43
2.3.1	Parameters	43
2.3.2	The Driver Routine	45
2.3.3	Growing a Configuration	47
CHAPTER 3	THE COST MODEL	51
3.1	OVERVIEW	51
3.2	INTEGRATED CIRCUIT COST MODEL	53
3.2.1	Chip Cost	53
3.2.2	Packaging Cost	60
3.2.3	Power	62
3.3	NON-ELECTRONIC COMPONENTS	63
3.4	SUBSYSTEMS	66
CHAPTER 4	TRIAL ARCHITECTURES	74
4.1	INTRODUCTION	74
4.1.1	Architectures Considered	74
4.1.2	Assumed Load	82
4.1.3	Default Parameters	88
4.2	PAGING ARCHITECTURES	90
4.2.1	Disk	90
4.2.2	Magnetic Bubble Memory	97
4.2.3	CCD Memory	100

4.2.4	Electron Beam Accessed Memory	102
4.3	SERIAL ASSOCIATIVE ARCHITECTURES	105
4.3.1	Logic-Per-Head Disk	105
4.3.2	Bubble and CCD Logic-Per-Track	112
4.4	DISTRIBUTIVE FUNCTION ARCHITECTURES	117
4.5	ONE-LEVEL STORE ARCHITECTURES	128
4.5.1	RAM	128
4.5.2	EBAM	131
CHAPTER 5	DISCUSSION OF THE RESULTS	133
5.1	INTRODUCTION	133
5.2	PARSING IS NOT THE PROBLEM	134
5.3	USER SHARING ALTERNATIVES	139
5.4	MEMORY TECHNOLOGY	143
5.5	ARCHIECTURES WITH DISK	149
5.6	IMPLICATIONS OF DISTRIBUTED FUNCTION	155
5.7	DIRECTIONS FOR FUTURE RESEARCH	159
CHAPTER 6	REFERENCES AND BIBLIOGRAPHY	162

APPENDIXES

<u>Appendix</u>		<u>Page</u>
APPENDIX A	EXAMPLE OF THE MODEL	A- 1
APPENDIX B	DETAIL OUTPUT SAMPLE	B- 1
APPENDIX C	COST SUMMARIES OF THE CASES	C- 1

ILLUSTRATIONS

<u>Figure</u>		<u>Page</u>
1-1	Moore's Law	3
1-2	Trends in Data Management Systems	5
1-3	High- vs. Low- Level Languages	10
2-1	The Configuration Model	27
2-2	Hierarchy of Levels of Abstraction	29
2-3	Basic Node Structure	31
2-4	Dispatching Disciplines	34
2-5	Architect's View	45
3-1	Mature Chip Cost Extrapolation	55
3-2	Maturity Learning Curve Factor	57
3-3	Volume Learning Curve Factor	58
3-4	Overhead Factors for Support Chips	59
3-5	Disk Drive Capacity	65
3-6	Subsystem Cost Calculations	70
4-1	User Sharing Alternatives	81
4-2	Paging Architecture	90
4-3	Logic Per Track Architecture	108
4-4	Distributed Function Architecture	120
5-1	Relative Cost of Sharing Alternatives	141
5-2	Network Bandwidth Requirements	142
5-3	Memory Technology Cost Forecast	143
5-4	Page Transfer Time	145
5-5	Time to Transfer a Megabyte	146
5-6	Effect of Memory Technology	147
5-7	Logic Per Head vs. Distributed Function	153
5-8	Optimal Disk Architectures	155

CHAPTER 1 STATEMENT OF THE PROBLEM

1.1 BACKGROUND

The development of the computer in the twentieth century has brought about a second "industrial revolution" in the way mankind conducts its everyday affairs. Just as the steam engine freed man from back-breaking physical labor, the second industrial revolution of the computer age promises to free man from mind-dulling routine, freeing more human potential for creative thinking to advance both intellectual and economic activity.

The short history of the computer industry can be viewed as a struggle to fulfill the promise of the second industrial revolution by making the power of the computer accessible to an ever-larger fraction of the population. In order for this to happen, computer hardware has dramatically declined in cost while increasing in speed. Supporting software has grown more complex and more capable. At the same time, the languages by which humans make their intentions known to the machine have improved.

Each of these developments can be classified into "generations." The first generation of commercial computers is usually thought of as the years 1955 to 1958, characterized by vacuum tube computers. Transistorized computers launched the

second generation, usually ascribed to the years 1959 to about 1966. With the third generation, integrated circuits became the dominant means for implementing computers.

As shown by Figure 1-1, the complexity of state-of-the-art integrated circuits has grown exponentially. (*) This growth in functionality on silicon drives the computer industry into a contemplated fourth generation of computer hardware.

(*) The first industrial revolution provided perhaps two or three orders of magnitude growth in the mechanical energy available to human beings over a period of one hundred years. It is sobering to contemplate the prospect of seven orders of magnitude growth in just thirty years from the silicon revolution.

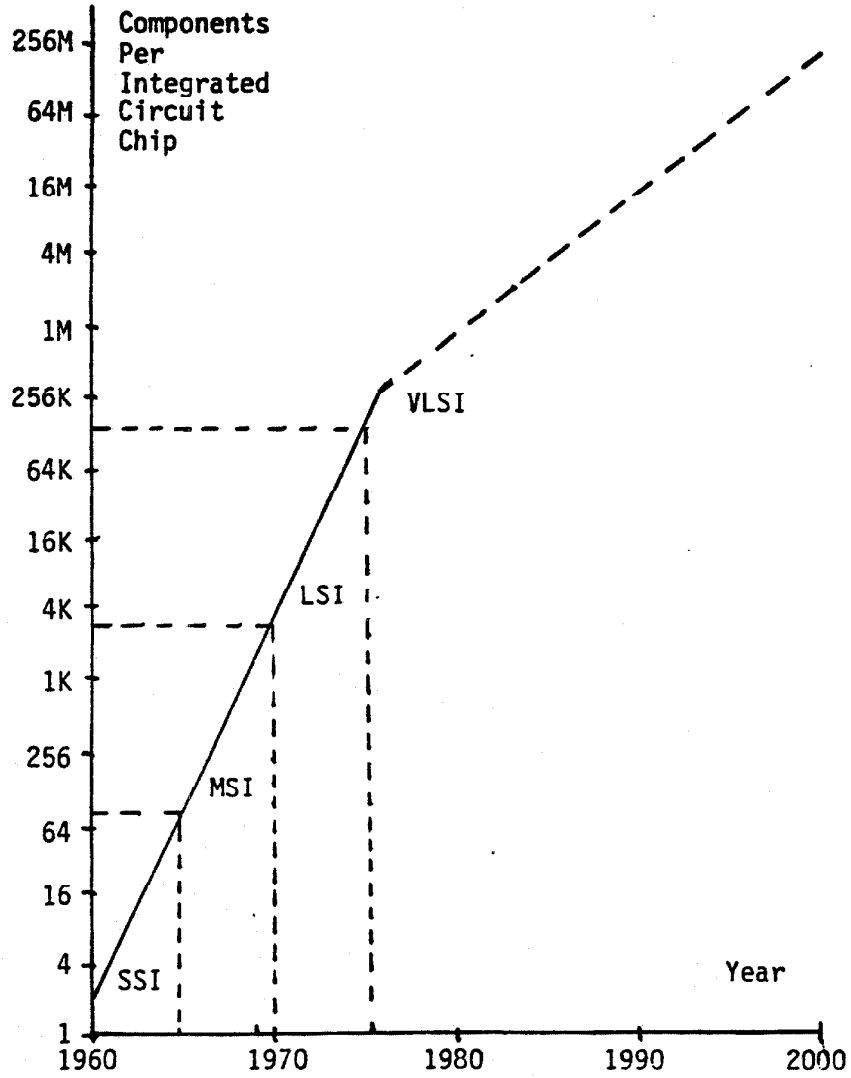


Figure 1-1. Moore's Law
Progress in Silicon Technology
(after [Noyce77])

The hardware revolution provides not only faster versions of existing computers and memories, but also the prospect of new

kinds of devices and new fundamental problems in system design. Sutherland and Mead eloquently argue this point [Sutherland77], pointing out the rising relative cost of interconnection and sheer magnitude of the design task as the critical challenges facing hardware designers in the 1980's. At the same time, however, the opportunity to depart from conventional computer architecture lies inherent in the burgeoning technology.

As computer hardware has become more capable and less expensive, the software supporting operation of the computer has also grown apace. Figure 1-2 summarizes some of the historical trends in operating software as they affect the problem of management of data, which historically emerged early as the primary use of computers.

Figure 1-2. Trends in Data Management Systems

First Generation

Rudimentary i/o software,
"hand crafted" for each job.

Batch processing of sequential
files on cards or magnetic
tape.

Users given lengthy
print-outs.

Data only as recent as last
run (day, week, month) are
available.

Managers requesting a new
report often told they cannot
have it.

Once established, data base
structures almost impossible
to change; application
program depends on nature of
the physical storage device.

First introduced in mid-1950's;
example: any COBOL job shop.

Second Generation

File storage software
with "access methods."

Mostly batch processing
with some on-line (disk)
files.

Users given "exception
reports."

Data valid as of when
the disk was loaded,
usually start-of-day.

Managers requesting a new
report must wait for a
program development cycle
of weeks to months.

Operating system requests
convert application program
requests to hardware
operations; some reformatting
is possible.

First introduced in late-1950's;
example; airline reservation
systems.

Figure 1-2 (continued).

Trends in Data Management Systems

Third Generation

Fourth Generation

Data Base Management Systems (DBMS) software packages.

Advanced information systems.

Transaction-oriented on-line systems; both batch and realtime applications; simple on-line functions only.

System is totally interactive; almost all requests are made on-line.

Users given terminal inquiry for only certain pre-programmed functions.

Users can search the data base and generate unanticipated, tailored reports.

Data maintained in real time; periodic batch reorganization needed to maintain integrity.

Data maintained in real time.

Request for a new report takes days to months, depending on how well request fits existing data base structures.

Requests for new reports on existing information answered on the terminal with system response time of seconds.

Data base structures can be changed without recompiling applications programs; but structure change takes weeks to months to implement.

Data structures manipulated interactively.

Date from mid-1960's; examples include IMS, CODASYL, etc.

Evolution towards these systems culminating in the late 1980's.

The decreasing price of hardware and increasing capability of software combine to make computerized information management systems available to an increasing spectrum of the population which provide increasingly facile ways to assist human beings in planning, managing, and accounting for their activities. The

state of the art has progressed greatly since the first generation of commercial computers. At present, systems can usually respond interactively to pre-programmed functions. Programs can be written under data base management systems without impacting the file requirements of other programs in the installation. Some capability exists for posing discretionary, unanticipated queries on-line, provided however that the data base is already properly organized to efficiently handle the request.

However, while systems have become extremely capable of managing rapid change in the contents of data bases, it is still difficult to modify the structure of a data base because control of data base structures resides in a central authority called the "data base administrator," who has only fairly awkward tools with which to manipulate the structure of the data base. In existing data base management systems, structural changes often require complete reorganization of the data base, thus retarding the ability of the organization which owns the system to make progress in its applications.

Structural change in a data base, however, cannot be avoided in a dynamically growing and changing organization [Thompson68]. The inability of present systems to adequately cope with this kind of change underlies the widely recognized problem of the

high cost of application software maintenance which plagues the industry today [Share76].

Further, the lack of truly high-level languages hampers the ability of the non-programmer to interact in more than just a mechanistic way with a data base. While some systems offer "query languages" which can in many cases express a complex request in a few lines, the user of the system is forced to program a majority of functions in a conventional algorithmic language such as COBOL either for efficiency of use of resources or because the full capabilities of the system are not available through the query language.

The fourth generation of "advanced data management" systems takes shape against the background of rapidly changing hardware capability. What will characterize these advanced data management systems? Such systems will have a natural user interface, the ability to define problem-domain specific languages, and distribute the control of the data base to end-users.

The term "natural user interface" means a very high level language, which could approach natural language in expressiveness, but only for a limited domain of discourse [Thompson63]:

The typical professional works in a narrow, highly technical context, a context in which many interlaced meanings have developed in the course of his own work. The clues that can be found in ordinary discourse are not sufficient to distinguish that highly specific context from all possible contexts. The idea that a single natural language processing system that will ingest all of the world's technical literature and then produce on demand the answer to a specific request may be an appealing one. But, at this time there are no useful insights [into] how context-delimiting algorithms would operate to meet the requirements of such a system. [Thompson75]

Within a limited domain, however, it will be possible to define problem-oriented, non-procedural, very high level languages which users can extend as required. A system of this sort will better fit the known tendency of human beings to express new knowledge and insight by moving from one recursive language to another [Randall70].

Problem-oriented languages employ constructs which fit the problem domain, as opposed to constructs which map onto the primitive operations of either the hardware or the operating system software. Under this definition, most contemporary computer languages are not problem oriented. To illustrate this distinction, consider a language for an inventory control application. In a problem-oriented language, stock items would be primitives of the language, whereas in a contemporary programming language, "stock item" would be expressed as a field in a record, for example.

Figure 1-3 amplifies on the notion of "high level" language as used in this context. By this standard, most existing programming languages are not high level, although some are quite far removed from the binary bits which comprise the machine program.

Figure 1-3. High- vs. Low- Level Languages

<u>Attribute to Compare</u>	<u>High Level</u>	<u>Low Level</u>
Length of the compiled code in terms of semantic primitives	short	long
Length of the execution path to interpret one rule of grammar	long - (routine)	short - (machine instr.)
Length of the statement the user must make to accomplish one unit of the user's work	short (sentence)	long (program)

The user must be able to extend the language. He should be able to introduce new vocabulary, and update both the intentional and extentional structure of the language. Extentional structure refers to data stored in the language version: the data themselves and any explicit relations among them. Intentional structure refers to concepts defined in terms of other vocabulary of the language (including intentional definitions) which in effect define new algorithms.

With these systems, the end-user can interact directly with the computer to satisfy most of his information needs. The role of the professional programmer will change from that of a necessary intervenor between the end-user and the computer system to that of an implementor of problem-specific languages which can solve the requirements of many end users. This role does not require the programmer to intervene in the end-user's job, which is currently a source of frustration not only to the end-user, but also to the programmer and the installation as well. [Share76] predicts that the evolution of the programmer into the "mid-user" of advanced systems will finally provide a satisfactory and professional environment to end-users, programmers, and installation managers.

1.2 THIS INVESTIGATION

Advanced data management systems will take shape in an era of great change in underlying hardware technology. The silicon revolution will provide the raw power to make these systems not only technologically possible, but also economically justifiable to ever-broader classes of users.

Advancing hardware technology introduces many new choices for the designers of these advanced data management systems. While existing types of memories and processors continue to improve in performance and decrease in cost, the silicon revolution brings new trade-offs in the forms of changed cost of interconnect and new types of devices. Many workers have proposed forming these elements into novel architectures for "data base machines."

What will the impact of these hardware developments be on the design of computer systems to support advanced data management? How will the new types of memory fit in, if at all? Do the parallel and associative processing schemes widely discussed in the literature offer an economic advantage as well as a performance advantage? If the intelligence of such a machine is to be distributed among many processing elements, how much intelligence should each unit possess?

None of the design trade-off decisions which affect these systems can be made in a vacuum: they strongly depend not only on the cost and performance of the hardware subsystems, but also on the pattern of use applied to them by the organization of people which will own the system. This work explores the interaction of the demands of a user organization and the architecture of an advanced data management hardware solution to elucidate some of these very high level design trade-offs.

The investigation considers the capabilities of the REL prototype system, as developed by Professors Frederick and Bozena Thompson and their students, as a fixed starting point which defines the capabilities of the class of advanced data management systems to be studied. The REL prototype exhibits many of the characteristics of fourth-generation data management systems discussed in the previous section: a natural user interface, (*) the ability to support many problem-domain-specific languages, (**) and distribution of control over data bases. (***)

The task of studying interactions of user organization demand on a variety of hardware architectures required a unique modeling system. The model described in Chapter Two builds upon the established technique of analytic queueing network modeling by applying both performance goal constraints and heuristic configuration grow procedures. The resulting model permits facile description of an architecture as a collection of hardware subsystems where the exact number and speed of each subsystem is parameterized. The hardware architecture is modeled in terms of

(*) REL ENGLISH. See [Thompson78] for user's guide documentation.

(**) For example, the REL Animated Film Language [Thompson 74c]. Extensions to the capability for defining new languages are discussed by [Solovits73] and [Hess80].

(***) See [Yu80].

the highest level of system description, adapted from Bell and Newell's PMS language.

The described architecture can be subjected to a range of load conditions, simulating different sizes and kinds of user organizations. The model will find configurations or instances of the architecture which meet the performance constraints at minimum cost by applying the heuristic grow procedures, and report the cost of the resulting configuration.

Because the model deals in exact numbers and speeds of hardware subsystems, the cost functions are calculated with relatively good precision not only in the orders-of-magnitude, but also in the coefficients. In previous work on advanced data management architectures, costs have been treated only in terms of orders-of-magnitude, with a few numeric examples for specific operations. This work permits comparison of expected dollar costs for these architectures in the context of organizations which will be expected to buy these systems.

In order to develop the cost coefficients used by the model, it was necessary to make a forecast of the costs of the various hardware subsystems. This forecast drew heavily on the technology forecasting literature for the semiconductor industry, and is discussed in Chapter Three.

To explore some of the range of possibilities posed by the hardware arena, thirty trial architectures incorporating a range of new and existing types of hardware elements were modeled using the tools described above. Chapter Four describes the architectures and how each was modeled. In all of these cases, an identical set of loads was imposed, representing organizations ranging from 1 to 500 users with processing demands ranging from trivial to tremendous. Chapter Five discusses the trade-offs illuminated by this set of architecture case studies.

CHAPTER 2 THE MODEL

2.1 INTRODUCTION

2.1.1 Overview

How well can a given architecture satisfy the requirements of a community of users? The answer to this question requires information which spans several views of the requirements and capabilities of a configuration of logic and memory elements. The PMS-Level Queueing Network Model ("the model" for short) integrates these viewpoints in a coherent structure which provides results which in turn permit comparison of architectures in a meaningful way.

What are the relevant views of this problem arena?

At one extreme, consider the organization's view of the system. In this realm, the system is thought of as supporting various classes of users, each with a set of functions to perform. These functions are called "transactions" in this work. The manager or operations researcher must quantify such things as the response time objectives of the system, the number of users of each user type, rates of requests, etc., to characterize this

view to the model. The organization view gives the load of work imposed on the system by its users.

At the other extreme, the hardware view of the system considers the subsystems (memories, microprocessors, disks, busses, etc.) which make up the physical configuration of the system. The hardware view is expressed in terms of the performance of the subsystems: the instruction rate of a processor or the access time of a disk, for example.

These views are reconciled by the system analyst's view. The system analyst view decomposes the transactions into a set of flows of activity. The flows are further decomposed until they can be represented by steps. A step is the use of the resources of one of the hardware subsystems (nodes) in order to accomplish part of the processing of a flow.

Finally, the model accounts for the architect's view of the configuration. Here, the term architecture refers to a configuration of hardware subsystems (microprocessors, disks, bubble memories, busses, etc.) together with the specification of the flows through the configuration. However, an architecture does not include the number or size of these various subsystems (e.g., the number of disks, the speed of the microprocessor, or the size of a certain type of memory). The terms case and

configuration refer to a particular instance of an architecture where the numbers and sizes of all of the subsystems are specified. Thus an architecture can be considered to be a family of cases.

The architect is concerned with describing the family; how one specific configuration can grow to become another; what are the limitations on such growth; what is the cost of a configuration; and how does the architecture fit a particular user organization, or a range of such organizations?

2.1.2 Modeling Technique

The model views the configuration as a queueing network. The nodes of the network correspond to the hardware subsystems of the configuration. Transactions are modeled as flows of activity through the queueing network, stopping at specified nodes for service. Each node is modeled by simple analytic formulae; times for the flow at each node are accumulated by linear combination.

The theoretical validity of the analytic queueing network technique follows from Jackson's Theorem [Jackson57], which holds that in a queueing network of nodes with exponentially distributed service times, even with feedback from one node to

another, the system behaves as if arrivals at each node were Poisson distributed (although they are not, in general). If $p(k[1], k[2], \dots, k[n])$ denotes the probability of $k[i]$ customers (*) waiting at node i , then Jackson showed that

$$p(k[1], k[2], \dots, k[n]) =$$

$p[1](k[1]) * p[2](k[2]) * \dots * p[n](k[n])$ where $p[i](k[i])$ is the solution for node i to the ordinary M/M/m queueing problem (*) (the probability of $k[i]$ customers waiting, where there are $m[i]$ servers, $\lambda[i]$ arrival rate, and $\mu[i]$ service rate (**)). Thus, Jackson's Theorem means that queues at each node can be calculated independently and combined linearly.

(*) In queueing theory, the term "customer" means any request waiting for service at a node.

(*) Queueing theory uses a short-hand notation of the form "a/b/c[/d[/e]]" form for describing the type of queueing situation. The first items, "a" and "b" refer to the distributions of request arrival times and service times respectively. "M" stands for "Markovian" or exponentially distributed when used for these parameters.

"c" refers to the number of servers, and is either a number (such as "1") or a variable (such as "m").

The last two fields of the notation are optional. "d," if present, is the maximum number of requests which can be in the queue. If absent, this field defaults to infinity.

"e," the final field, is the queueing discipline, which can take on values such as FIFO (First In First Out), LCFS (Last Come First Served), SJF (Shortest Job First), etc. If absent, FIFO discipline without priorities is assumed.

(**) These terms are used with standard meanings in queueing theory. See [Kleinrock75] or any general text on queueing theory for precise definitions.

C. G. Moore introduced analytic queueing network models to the computer systems analysis domain in his work on large-scale timesharing systems [Moore71]. This author has successfully applied queueing network models to analysis of a major computer/communications network consisting of over 60 computers; 4,000 communications lines; and 14,000 terminals [Neches76]. Both studies reported agreement between model predictions and measurements of actual performance of better than 10%.

Analytic queueing network models have several advantages over discrete event simulation models for predicting the behavior of a large and complex system. The analytic model is easier to write and debug, and is also much more efficient of computer run time. This permits its use in studies where many architectures must be described, and a large number of cases run on each.

In addition, the analytic model lends itself to extrapolating the behavior of a large number of similar resources from the behavior of a single resource. Such extrapolation saves both computer time and effort to describe the system to be modeled. In discrete event models, it is usually necessary to represent each resource to be modeled individually (at least at model execution time).

The compactness of description, aggregation of similar resources, and run-time efficiency of the analytic model made it possible to deal with the range of configurations and number of cases run against each configuration in this thesis. Discrete event simulation was abandoned early in this research [Neches78] primarily because each configuration would have required writing a new simulation program and secondarily because of the amount of computer time which would have been required to run a meaningful set of cases.

The analytic queueing network technique is based on the assumption that the nodes are independent, so that their response times can be combined linearly. An equivalent statement is that there exists infinite storage capacity in each node for incoming requests.

Experience with analytic queueing network models shows that the assumption of independence is usually not a significant source of error as long as no node in the network is close to overload. In a network which includes one or more overloaded nodes, the analytic model fails to account for the fact that congestion spreads from the finite queue of the overloaded node to its neighbors in the network. This spreading congestion adds to the workload of the neighboring nodes, perhaps causing them in turn to become overloaded. It is just this kind of dependent

behavior that accounts for the extremely long response time delays encountered, for example, in a timesharing system undergoing "trashing" due to lack of a resource such as main memory. The analytic model will, however, pinpoint the node responsible for the situation [Gordon67].

Fortunately, the model as used in this research is guaranteed to keep all nodes in a given configuration from becoming overloaded, thus avoiding the problem of the accuracy of the simulation technique just discussed. The architectural view portion of the model does this by growing the capacity of any hardware subsystems (nodes) which become response-time critical.

More sophisticated analytic queueing network models have been devised which can handle non-exponential servers (*) [Muntz72a], provided that the output of each node in the configuration can be shown to be Poisson distributed [Muntz72b]. These models unfortunately lead to very complex solutions; in practice the computation time can be quite high, even with well optimized algorithms [Muntz74]. Thus given the desire to run large numbers of cases, the restriction to only exponential servers and the resulting loss of generality did not seem too severe.

(*) Servers with constant, hyper-exponentially distributed, or queue-length dependent service times are examples of non-exponential servers.

2.1.3 Implementation

The model is implemented as a PL/I program running under the MVS operating system on the IBM 3032 processor at Caltech. Appendix A gives a full source listing of the model, as used to run one of the architectures discussed in Chapter 4.

Each run of the model is accomplished by concatenation five source files and then compiling, linking and executing the resultant PL/I program. The code for the model is contained in three of the files: these three files were used to make every run discussed in this thesis.

The first file, PLIMOD4A, contains data structure declarations and initialization code for this model. It appears on pages A-2 through A-4 and is comprised of statements with line numbers of the form 1XXXX.

The second file, PLIMOD4B, contains bridge code and declarations for the CONFIG routine. The CONFIG routine runs one specific case, with all parameters instantiated, thus modeling one instance of an architecture. The code from this file appears on pages A-10 and A-11, comprising line numbers of the form 3XXXX.

The third file, PLIMOD4C, contains the majority of the code for the model. The routines in this file use the data structures declared in the first file and the calls from the configuration file to build an internal representation of the case to be modeled. Then the analytic queueing network model is applied by the routines in procedure PERFORM_CALCULATIONS. The results are then available to be queried from the architecture view file. A report writer routine is included to provide a detailed output of the results of a particular case. Code for this file appears on pages A-16 through A-35, with line numbers of the form 5XXXX.

The fourth file is called a "configuration" file and defines the architecture to be modeled. This file consists of three sections: the hardware view (node declarations), system analyst view (flow declarations) and user organization view (transaction declarations). In general, a new configuration file was written for each architecture studied. However, some of the architectures were so closely related that they used the same configuration file, relying on parameters to differentiate between CCD and bubble memory characteristics, for example. Code for this file appears on pages A-11 through A-15, with line numbers of the form 4XXXX.

The fifth file is called a "driver" file, and contains the code which implements the architect's view. It describes which

nodes of the configuration are allowed to grow and how this growth is to take place. A new driver file was written for each architecture modeled. The driver file appears from page A-5 through A-9 in the example, and is comprised of statements with line number of the form 2XXXX.

The architecture view files and configuration files in effect provide a programming language for describing the simulations to be run. This language can be thought of as the capabilities of the model augmented by the expressive power of PL/I. Embedding specialized functions in a general purpose programming language can result in specialized language without the effort of writing a compiler. The construction of LAP on SIMULA67 provides another example of the use of this technique [Locanthi78].

An organization's view specified a load on an architecture. In running a case, if the average response time to users in the organization was above the specified minimum, the configuration grew and a new case was run. On the IBM 3032 at Caltech, a typical run of the model required 7 to 8 seconds of CPU time for compilation and linking. The model then required between 15 and 50 milliseconds of CPU time per case. A typical run of a given architecture required between 1,000 and 5,000 cases.

2.2 THE CONFIGURATION MODEL

The configuration model, or CONFIG procedure, is the heart of the simulation program. The CONFIG procedure consists of the configuration file supplied by the modeler and standard declarations and routines included in the overall program structure. Each time it is invoked, the CONFIG procedure defines one specific instance of the architecture to be modeled by instantiation of all hardware and user view parameters. The analytic queueing network model is executed against this configuration. Thus, the CONFIG procedure is the "inner loop" to the architecture view.

The configuration file consists of three sections, corresponding to three of the abstract views of the system (see Figure 2-1): Node definitions correspond to the hardware view; Flow definitions correspond to the system analyst view; and Transaction definitions correspond to the user organization view.

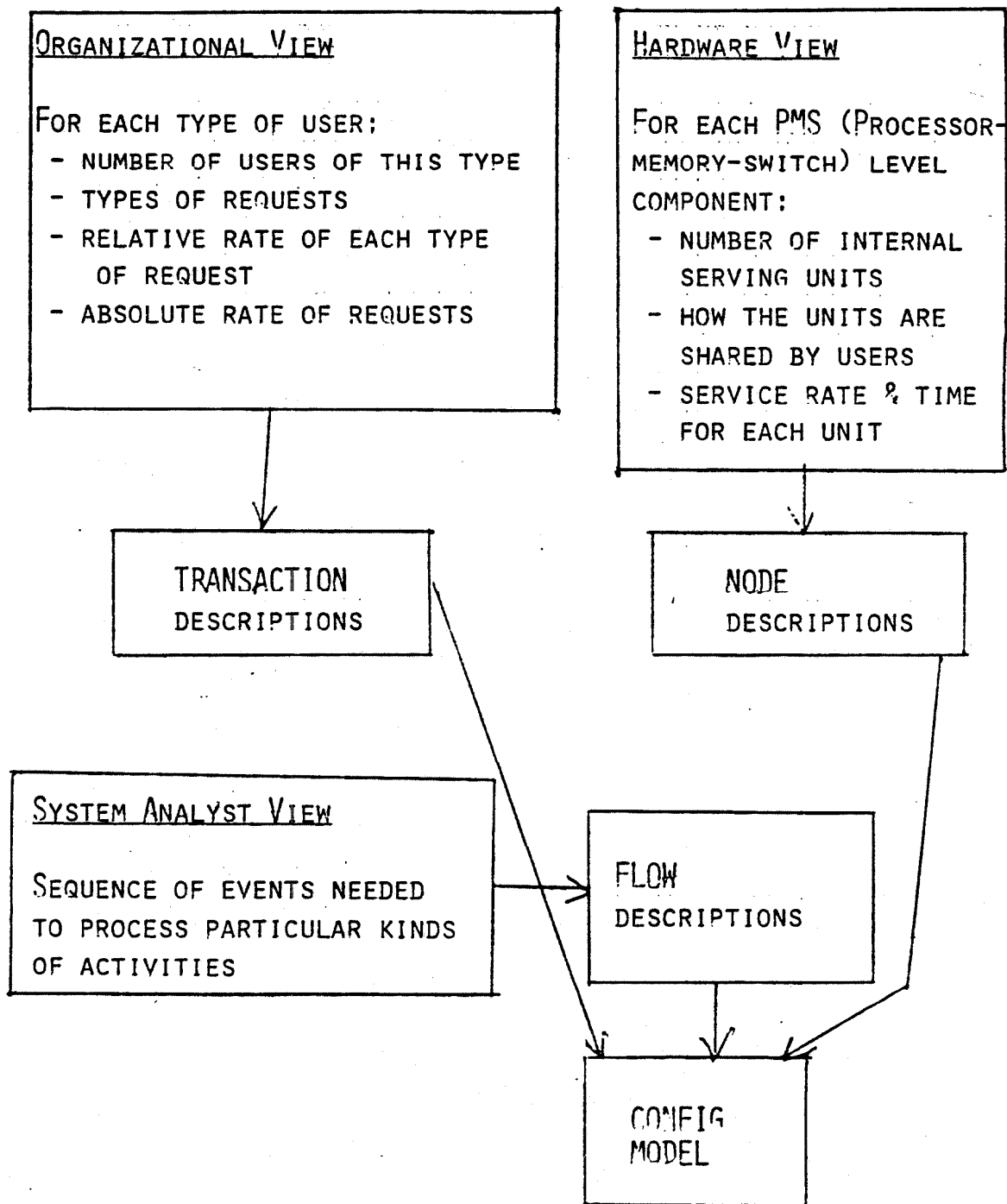


Figure 2-1. The Configuration Model

Each section makes calls on model routines to transmit these definitions to the data structures of the model. Because the file is a part of a PL/I program, the full capabilities of that language are available to the modeler to define parameters and do calculations on them which make the parameters more meaningful. It is also possible -- and quite useful -- to include modeler-written PL/I procedures to further simplify and compact the definition of an architecture.

2.2.1 The Hardware View

"Computer systems are one example of man's more complex artificial systems" [Bell71]. Figure 2-2, reproduced from Bell and Newell's classic text, illustrates how people have been able to cope with the complexity of computer systems by breaking the process of designing and understanding them down into several "levels of abstraction."

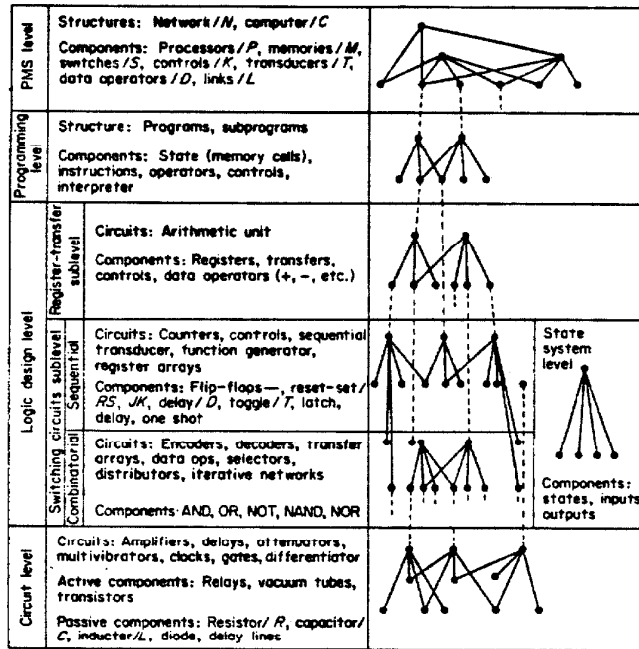


Figure 2-2. Hierarchy of Levels of Abstraction in Computer Structures [Bell71].

A system can be modeled at any of these levels of abstraction, depending on what information the designer seeks from the model. Here, we are concerned with comparing the behavior of entire classes of computer systems at a level which would be visible to the end-user. (*) Only the highest level of abstraction available for quantitative description will be useful in this endeavor. Otherwise the model would be so expansive in detail as to make the task of completing one configuration case

(*) That is to say, in terms of response time and dollar cost to meet a given load requirement.

study -- let alone an architecture study -- prohibitively time consuming.

Bell and Newell propose a descriptive language for this top level of abstraction called the "PMS" (Processor, Memory, Switch) level, and apply their language to discussing many of the significant computer architectures then extant. In this thesis, the PMS level is the natural level to employ in description of the architectures to be studied.

PMS, as presented by Bell and Newell, is not directly suitable for use as a simulation input language. The two-dimensional structure of PMS diagrams is hard to interpret algorithmically. PMS descriptions can contain both quantitative and qualitative terms. For modeling, the system description language must be consistent, linear, and quantitative. These are the goals of the node description mechanism of the model.

In the model, each PMS-level component is a node. Nodes do not distinguish the function of the component: the same kind of node is used to model processors, memories, busses, etc. The node concept abstracts out of the PMS component only those attributes which influence the service rate.

It proved useful to have a basic node for modeling with a somewhat more complex structure than just a single queue and serving element. Figure 2-3 shows the node structure, which is similar to [Jackson57]. Each node consists of a number of identical basic serving elements ("basic servers"), a queue, and a dispatching rule for sending requests from the head of the queue to basic servers.

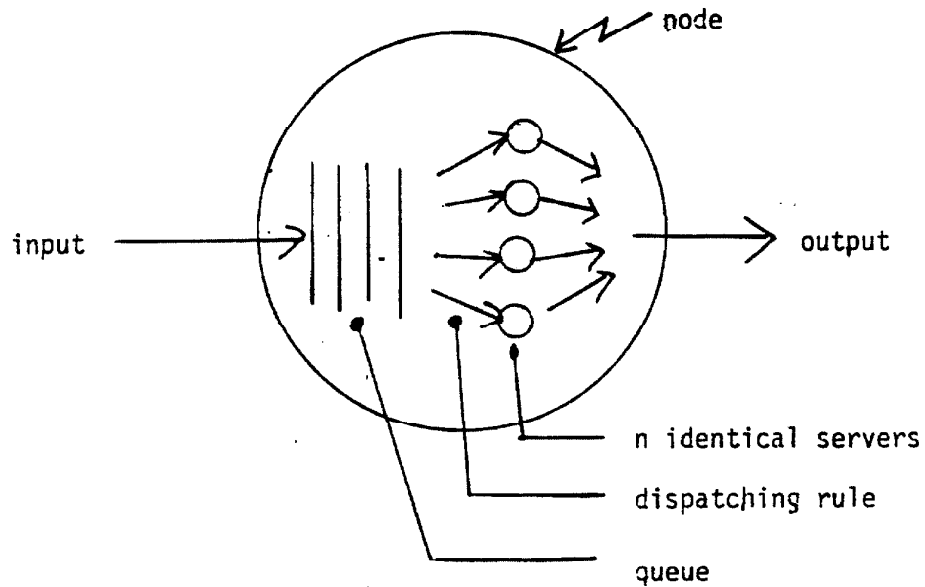


Figure 2-3. Basic Node Structure after [Kleinrock76], p. 215.

Two variables characterize the basic server. PAGESIZE gives the size, typically in bytes, of the smallest distinguishable and uninterruptable unit of work for the basic server. PAGESIZE can

be picked to reflect the nature of the hardware unit being modeled. The other variable, PAGERATE, gives the throughput of the basic server in PAGESIZE units per second.

For example, in modeling a disk drive, the seek element can be modeled with a PAGESIZE of 1 (seek) and a PAGERATE of 33 (seeks per second). This is done because the seek operation is normally uninterruptable. A processor is modeled with a PAGESIZE of 1 (instruction) and a PAGERATE of so many million (instructions per second). Another case might be a system bus, such as the DEC UNIBUS, which would be modeled with a PAGESIZE of 2 (*) and a PAGERATE of 2.5 million. (**)

A node in the model consists of one or more basic servers. The raw capacity of the node is thus SERVERS times the capacity of a basic server. With this capability, aggregates of similar resources can be described easily. For example, a string of disk drives can be modeled by a single node where the basic server has the throughput characteristics of a single drive, and SERVERS gives the number of drives in the string.

(*) because of the 16-bit wide data paths

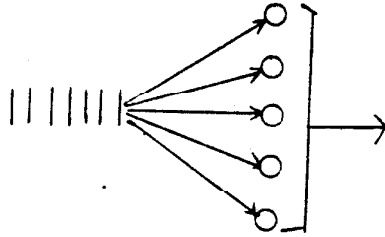
(**) the reciprocal of the bus's nominal 400 nanosecond cycle time.

Nodes have a single entry point for requests and a single exit point for responses. A request can be thought of as a token (in the Petri net sense) which advances through the internal queue until it is dispatched to a basic server. After spending an appropriate amount of time in the server, the pebble exits the node and enters the queue of the next node in its flow.

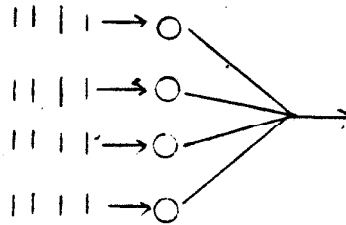
The dispatching rules describe how requests can move from the queue to servers. Each dispatching rule corresponds to one of the queueing systems describable by queueing theory: the model can accommodate any type of queueing system for which a closed form analytic solution exists for the mean and variance of expected service time. (*)

For the cases presented in this thesis, only three simple dispatching rules sufficed to describe all of the modeled situations. Admittedly, more general queueing models could have been used, but it is felt that the added sophistication was not justified in light of the uncertainties in the assumptions for node service times in many of the cases. Figure 2-4 shows the three dispatching rules implemented by the CALCULATE_QUEUEING procedure (listed on page A-25).

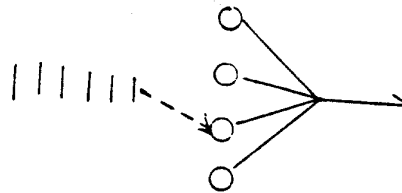
(*) Section 2.1.2 discusses other restrictions



(a) QUEUE TYPE=1: Parallel serves, M/M/1 queue.



(b) QUEUE TYPE=2: M/M/1 queue per server.



(c) QUEUE TYPE=3: M/M/n queueing system.

Figure 2-4. Dispatching Disciplines

QUEUETYPE=1 nodes have a single M/M/1 queue, but the request is dispatched to all servers simultaneously. Each server handles 1/Nth of the request (rounded up to the nearest PAGESIZE unit). This rule easily models interleaved memory organizations and proved extremely useful in compactly describing a variety of parallel processing structures for data base processors.

QUEUETYPE = 2 nodes have an array of individual M/M/1 queues, one per basic server. Requests enter one of the queues and proceed all of the way through the queue to its server, analogous to the checkout lines at a supermarket. Requests enter the queue for each server with equal probability. This dispatching discipline would be used, for example, to model a string of disk drives, where a request to access a particular record can be served only by the disk drive which holds that record and thus must service the request to completion. Other drives in the string may be used concurrently to handle requests for other users, however.

QUEUETYPE=3 nodes have a single M/M/n queue feeding all n servers. As with the preceding node type, once a request is dispatched to a basic server, that server processes it to completion. In this case, since a request can be handled by any server, so it goes to the first free server. This queueing discipline prevails in most banking institutions today. This mode models systems such as shared memory multiprocessors.

A node can be shared by a number of users with similar characteristics. The model multiplies the imposed load by USERS to arrive at the actual load of a node. Each node in a model can be given a different number of users to indicate the degree to which it is shared. For example, the terminal microprocessor in a desktop system would have USERS=1, because it is dedicated to one user. A cluster controller might have USERS=6, where an archival memory in the same configuration might have USERS=100. All of these nodes can coexist in the same run, and the correct queueing effects will be calculated for each.

In this manner, the analytic nature of the model makes it easier to describe a very large configuration which serves perhaps hundreds of users. Resources which are replicated, such as terminals, can be modeled by description of a single representative such resource. By manipulating the USERS parameter, the modeler can easily account for the aggregation of load from many such similar resources.

An example of a small set of node definitions occurs on page A-11. In this example, it is worth noting that calculations are performed on some of the parameters to obtain such items as PAGERATE. This permits the modeler to deal with parameters which are more meaningful: for example, SHIFTRATE is a natural description of bubble and CCD memories. The modeler specifies

the transformations on the parameters using the programming concepts of PL/I.

2.2.2 System Analyst View

The system analyst view connects the hardware nodes as just discussed with the user (transaction) view (next section). Presumably, the analyst would perform the analysis from user requirements towards the hardware description in a top-down manner by progressive refinement of transactions into flows, and flows into either constituent flows or steps. However, for convenience of execution, the description occurs in bottom-up order. Bottom-up order of execution permits the program to check for steps that use non-existent nodes and transactions that rely on non-existent flows.

The system analyst first can define PL/I procedures for producing fragments of flow routines. These PL/I procedures can be much more general and perform more interesting calculations than the transaction-flow-step formalism of the model, and thus reduce the effort of description. These procedures, when executed, produce calls to procedure NEWSTEP; other calculations can also be performed as side effects.

The analyst then defines the flows. A flow is a set of steps, where each step represents the use of one of the hardware resources of the system (NODE) to process a certain unit of work (SIZE) repeated a certain number of TIMES. (*) Flows are used to represent basic system tasks which are common to one or more transactions, such as parsing an input sentence or computing a projection operator.

Flows differ from general PL/I procedures in that the model automatically aggregates response times for flows. Flow times can provide the modeler with some insight as to where resources are consumed and delays generated. Extrapolations of system response times can often be made by careful study of flow time reports.

The set of flow routines usually constitutes the largest portion of the configuration file. Fortunately, given a set hardware view and user organization view, the flow routines are usually straightforward to write. For an example of a set of flow routines which includes PL/I sub-routines for producing flow fragments, see pages A-12 to A-14.

(*) Having both SIZE and TIMES as inputs to specify gives the modeler control over how standard deviations are aggregated: an activity with a small SIZE repeated many TIMES will result in a very small standard deviation, where an activity done one TIME with a large enough SIZE to result in the same total amount of work will have a larger standard deviation.

2.2.3 User Organization View

The user organization view, as translated into a set of transaction definitions, constitutes the final part of the configuration file. The user organization is thought of as one or more classes of users with a set of transactions for each class of user. Within each user class, each transaction is given a WEIGHT; the weights for a class should sum to 1. The weight of a transaction is the relative number of user-originated requests for the activity represented by that transaction.

A user group is also characterized by a number of users and an average request rate per user. Weights are given to transactions as though there were only one user in the group.

A transaction is then further characterized by the sequence of flows which compose it (just as a flow is composed of steps). Each flow can be repeated a number of times, which need not be an integer, since it represents the average number of repetitions. Obviously, repetition factors can be parameterized. (*)

(*) This technique was used for runs in Chapter 4 to account for various degrees of interversion communication.

The model accumulates response times for each transaction. When a wide range of repetition factors is expected, the notion of "average repetition factor" will lose the sense of the distribution of response times (and most likely significantly underestimate the variance). In this situation, it is best to define a set of transactions which are identical in all respects except weight and repetition factor, and manually reconstruct the distribution of response time. (*)

In the cases discussed in Chapter Four, the set of transactions and weights remained constant for every architecture, thus subjecting each to identical user requirements.

2.2.4 Detail Output

The model contains a report writer routine (pages A-32 to A-35) which outputs in detail the results of the analytic queueing network calculations. The REPORT routine produces several output reports on each case; because of the volume of

(*) This can be done using a PL/I subroutine to generate the transaction by issuing a series of calls to USEFLOW. It never proved necessary to use this technique in any of the cases reported on here.

output generated, the report writer acts only when explicitly called from the driver (architecture view) file. Generally, the detailed report can be useful for understanding the interaction of the load with the configuration, but provides too much detail for architectural comparison. The driver files written for this thesis produce a detailed report only for a representative case out of each architecture study. Appendix B contains a sample of the detailed output, illustrating each of the reports.

The first report shows properties and calculated results by node (see page B-2). The columns headed "PAGE SIZE," "PAGE RATE," "USERS," and "SERVERS" echo the input to the queueing model. The remaining columns display calculated results.

"CAPACITY" is the total capacity of the node in PAGESIZE units per second.

"UTIL" is the utilization of the node across all basic server elements for the entire user load. (Utilization is usually represented by the Greek letter "rho" in queueing theory.)

"MEAN/SO" gives the ratio of calculated mean service time of the mode (including time waiting in queue) to the unloaded service time. (*) Similarly, "SIGMA/SO" is the ratio of the standard deviation of service time, including waiting time, to the unloaded service time. These normalized results give a clearer picture of congestion developing at a node than unnormalized values.

"WGTD TIME" is the amount of time, in seconds, the node contributes to the average transaction (weighted by the volume of requests of each type). The node with the highest weighted time is most likely to be the critical node, although this is not always the case.

The second report (page B-3) breaks down utilization by user transaction for each node. Utilization greater than 1 indicates an overloaded node: the response times calculated by the model will be meaningless.

The third report (page B-4) gives details on the flows. Each step of the flow is summarized on one line, giving the node visited, resources demanded, and the resulting mean and standard

(*) Unloaded service time is the time the request would take if it were the only request in the system - that is, if there were no other load.

deviation of response time. At the end of each flow, the mean and standard deviation of service time for the entire flow are printed. The flow report is useful for comparing designs because the flows represent common system functions, and it is often straightforward to extrapolate from them.

The final report (page B-6) gives a similar breakdown for transactions. Flow details for each transaction are printed, one flow per line. Mean and standard deviation of response time for the entire transaction appear after the flow details.

2.3 ARCHITECTURE STUDIES

2.3.1 Parameters

By defining a configuration in terms of parameters, an architecture can be modeled as the set of configurations possible with a range of values for each parameter. The architectural view, then, concerns itself with utilizing this parameterized configuration model by manipulating some of the parameters and observing the consequences on system cost and performance. There are four kinds of parameters:

(1) Some of the user organization parameters are forced across a range of values to represent a range of user requirement situations. Thus the architecture can be evaluated for suitability to a range of user organizations.

(2) Other user organization parameters are held fixed for the duration of the run.

(3) Some of the hardware view parameters respond to growth in the user load, driven by algorithms which permit them to change and grow in order to meet response time objectives.

(4) The remaining hardware parameters remain fixed by the choice of the architect.

The DEFAULT procedure sets all parameters to starting values. It can be invoked at any time to restore baseline conditions during a run of a series of cases. In general, the default case should be the smallest configuration which is a meaningful case of the architecture in question. An example of a DEFAULT routine appears on page A-7.

2.3.2 The Driver Routine

Figure 2-5 shows the overall structure of the architectural view in relation to the other views of the model. In a run, one or more of the user organization parameters will be forced through a set of values. For each distinct set of user organization parameters, the CONFIG (analytic queueing network) model will be run iteratively until the configuration either meets the response time criteria or cannot be expanded any further.

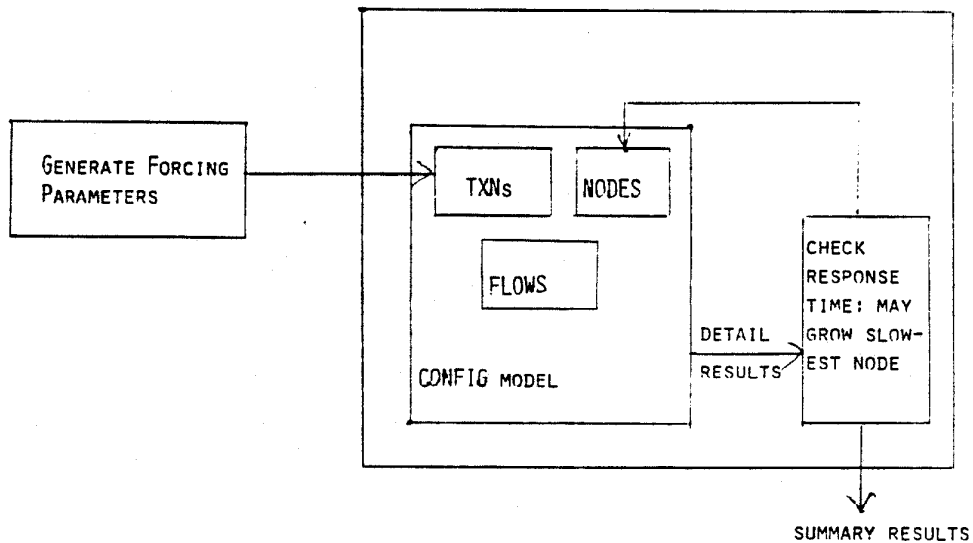


Figure 2-5. Architect's View

After each iteration of the queueing network model, the architecture view checks for convergence to the response time criterion programmed by the modeler. (*) The model then searches for the node which contributes the largest amount to weighted mean response time, and calls a modeler-written heuristic routine to grow the capacity of that node. If a maximum iteration count is not exceeded and the grow routine succeeded, the model iterates again through the queueing network calculation.

When the process converges, a modeler-specified line of output can be generated. This line usually contains values of hardware parameters which can be algorithmically grown in order to validate the reasonableness of the heuristics. This also gives the modeler a feeling for how the architecture responds to different load situations.

For configurations which converge, the model also computes a cost function written by the modeler. The costs are printed in a matrix at the end of each set of cases. For cases in which even the smallest configuration of the architecture exceeds the

(*) In this thesis, the response time criterion used is weighted average response time -- that is, weight (relative frequency) of each transaction in the mix times response time of that transaction. The time must be less than a constant arbitrarily set at 15 seconds, intended to represent the "frustration limit" of a human interactive user.

minimum required response time, the model assumes that the architecture is "TOO BIG" for the case, and probably far too expensive. Conversely, when the model cannot converge on a configuration which meets the maximum response time permitted, the model assumes that the architecture is "TOO SMALL."

The "driver" file expresses the architect's view and includes the GROW and COST routines. For an example of a driver file, see pages A-5 to A-9.

The driver file can also be used to drive some of the hardware parameters, while letting other hardware parameters respond via the GROW routine. This technique provides a particularly valuable way to study design trade-offs within an architecture, because the "bottom line" conditions of performance and cost, as seen by the end user, are the outputs.

2.3.3 Growing a Configuration

The GROW routine written by the modeler provides the mechanism for studying a range of configurations of increasing load demand. The GROW routine accepts an integer as input which gives the number of the worst-case node. The routine returns a boolean which indicates whether or not it was possible to grow

the configuration to enhance the capacity of the node in question: if not, then the architecture has reached a fundamental limitation. A typical GROW routine appears on page A-8.

The GROW procedure begins with a dispatch (PL/I computed GO TO) on the node to be expanded. Several different nodes can use the same algorithm for expansion in this way.

The modeler specifies how to add capacity to a particular node by writing algorithms which increase one or more of the hardware view parameters. The expansion algorithms can thus be defined in terms natural to the architecture in question. For example, a string of disk drives grows by adding one drive at a time to the string, where solid-state memories can best be increased in throughput by increasing the degree of interleaving.

Adding capacity to a particular node may involve changing parameters which also affect other nodes. For example, adding a disk drive to a string may result in exceeding the number of drives which can be handled by a controller. Adding processing elements in a bus-coupled architecture may increase the bus's electrical length, thus increasing the bus's cycle time and reducing its capacity. Effects like these can easily be reflected by grow routines.

The desirability of coding grow routines as architecture-dependent heuristics was not immediately obvious. A first attempt at the problem used a single EXPAND algorithm (*) which could be applied to any node in a general fashion. The EXPAND routine quickly proved to be too limited in that it would not account for the fact that some resources (such as disk drives) grow by discontinuous jumps (whole units in this case). The general routine could also not apply constraints of the form where growth of one resource required additional resources to be added (such as when adding a disk also forces adding a controller).

A final deficiency of the general algorithm lies in its inability to make "reasonableness" tests to see if further expansion of the node is either futile or would violate some other constraint. With the architecture-specific heuristic algorithms, however, such tests can be very straightforward. An example of such a test would be to see if no further outboard peripherals can be added because their aggregate transfer rates would exceed the bandwidth of the main memory.

(*) which survives in the program as a utility available to the heuristics (page A-10).

Some of the reasonable tests can be even more subtle. In some cases, intermediate results of the queueing model were queried by grow routines to determine if any amount of expansion of the affected node would be sufficient to meet the response time objective, independent of queueing delays.

Use of heuristic grow routines thus captures the constraints of the architecture very realistically. They have the added benefit of causing individual configuration cases to converge in very few iterations, thus saving considerable computation time over a more general algorithm.

CHAPTER 3 THE COST MODEL

3.1 OVERVIEW

The architecture view produces configurations which, although vastly different in underlying architecture, have roughly similar performance for a given set of user requirements. This with performance fixed by user specification, the obvious way to compare architectures is to compare costs.

The cost models used here aspire to show that cost in terms roughly similar to the unit manufacturing cost (UMC), in dollars, which a fully vertically integrated computer manufacturer might expect. Unit manufacturing cost means the factory cost of production: materials, labor, testing, and factory overhead. It does not include other costs which a company must cover in the selling price of the product such as cost of marketing, administrative expense, or profit. UMC gives a fairer comparison of different technologies than selling price because it is not as biased by marketing strategy (OEM vs. end user, for example), nor does it reflect prices which result from having an exclusive market position. UMC also begs the question of development costs.

Obviously, this kind of comparison has its pitfalls, as companies guard their actual UMC data jealously, if they have that data at all. Fortunately, in the case of integrated circuits, a wide literature exists projecting trends in the technology. (*) The cost information in that literature forms the historical basis for the model of integrated circuit chip costs used here.

Other cost estimates can be made by extrapolation of historical trends, as was necessary for disk drives. For proposed new technologies, such as EBAM or optical video disk, there was no recourse but to accept published cost estimates at face value.

In all of the cases presented in this thesis, the cost procedures as programmed are quite simple, usually written in terms of cost coefficients for major hardware nodes. An example of a COST procedure appears on page A-9. The following sections explain the rationale used to develop many of the cost coefficients common to many of the architectures found throughout this work.

(*) A good summary of this literature is [Early78], which gives fairly detailed cost forecasts for memory products through 1986. [Mohsen79] covers the 1985 to 1990 time period, but with less quantitative forecasting.

3.2 INTEGRATED CIRCUIT COST MODEL

The cost of a subsystem or assembly consisting primarily of integrated circuits can be thought of as the sum of three costs: the cost of the chips (made proportional to the number of chips in the system), the cost of packaging (made proportional to the number of pins), and the cost of power and cooling (made proportional to the power consumption in watts).

3.2.1 Chip Cost

The cost of a particular integrated circuit depends on many factors: die size, yield, process complexity, and "learning curve" (*) phenomena. A simple model to account for these factors might be:

$$\text{COST(chips)} = \text{NCHIPS} * \text{MATURE_COST} * \text{MLCF} * \text{VLCF} * \text{OVERHEAD(chips)}$$

where

NCHIPS is the number of primary type chips in the system. For example, in a memory, this would be only the number of memory chips.

(*) the ability to produce the same functionality chip at lower cost in succeeding years through design and process improvements and economies of scale.

MATURE_COST is the cost of manufacturing and testing a primary type chip after all learning curve phenomena have taken place: that is, the ultimate cost of the chip.

MLCF is a "maturity" learning curve factor which accounts for improvements in technique for making a given kind of chip over time.

VLCF is a "volume" learning curve factor which accounts for economies of scale in manufacturing.

OVERHEAD is a factor which accounts for the cost of non-primary type or support chips.

With all of the learning curve phenomena removed, the mature cost of producing chips of equal complexity but different functionality should be the same. Thus, this model assumes that if produced in large enough volume and for a long enough time, RAM, CCD, microprocessor, custom, and gate array chips with similar feature sizes should cost the same to produce because they are made by comparable processes.

The best quantitative data on cost trends for LSI and VLSI circuits can be found for dynamic RAMs. Figure 3-1 presents a view of mature cost trends extrapolated from the famous memory

cost graph of [Noyce77]. It is interesting to note that while RAM densities increase by a factor of four, chip costs increase by a factor of two. (*)

Figure 3-1. Mature Chip Cost Extrapolation

<u>RAM Density (bits)</u>	<u>Year Introduced</u>	<u>Year Mature</u>	<u>Mature cost cents/bit</u>	<u>Mature cost \$/chip</u>
256	1969	1975	.300 (est)	\$ 0.77
1K	1973	1979	.150 [Noyce77]	1.54
4K	1975	1981	.048 [Noyce77]	1.97
16K	1976	1982	.025 [Noyce77]	4.10
64K	1979	1985	.013 [Noyce77]	8.52
256K	1981	1987	.008 (est)	20.00
1M	1983	1989	.004 (est)	40.00

The tendency for chips to become more expensive reflects several major trends in VLSI technology. While advances in lithography reduce the size of features on the chip, active devices are shrinking faster than wires [Early78]. At the same time, the increased level of integration on a chip suggests the

(*) Memory system costs actually decrease by more than a factor of 2 with each quadrupling of RAM density because the decreasing number of pins and lower power consumption leads to lower packaging costs.

need for more interconnect both on- and off- chip [Keyes78]. Both of these factors suggest that an increasing percentage of the area of a chip will be devoted to interconnect, cutting in to the density improvements available from lithography.

The interconnect problem is recognized as the crucial design issue for VLSI [Mead79], and several strategies have emerged for coping with it. Increasing the regularity of the design, trading internal state for external interconnect, and increasing the number of levels of design hierarchy on the chip have all been suggested. The latter two techniques, however, have some cost in real estate on the chip.

Thus, despite advances in lithographic technique, average chip die sizes can be expected to increase [Pashley78]. As lithography pushes to finer resolutions, processing equipment tends to become more sophisticated and expensive, and sometimes lower in throughput (as is the case with direct electron beam exposure). The processes are also becoming more complex, as more masking steps are added. Finally, with larger scales of integration, the cost of testing becomes increasingly important, perhaps to the point that the circuits proposed for the middle and late 1980's will not be testable unless designed with more levels of design hierarchy on chip, which has a cost in area.

The chip cost equation includes two learning curve factors, one for volume and one for the maturity of the design. The maturity learning curve used here was derived from [Noyce77] and represents the average of his predictions. Figure 3-2 illustrates the maturity learning curve, which accounts for phenomena such as process refinements and scaled parts of the same design but smaller die size, and tighter control, all of which reduce cost by increasing yield.

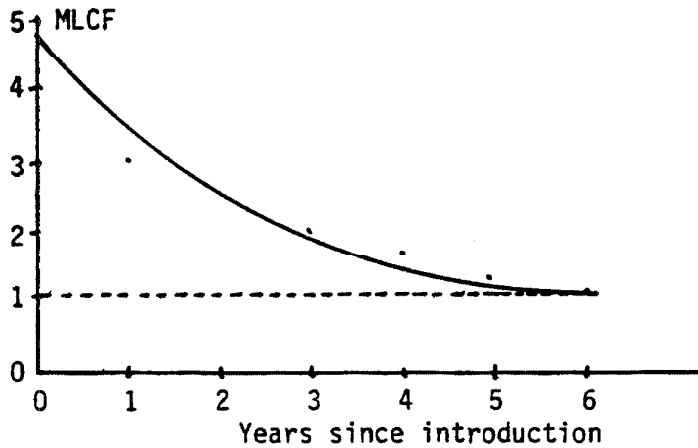


Figure 3-2. Maturity Learning Curve Factor

The volume learning curve shown in Figure 3-3 accounts for economies of scale in manufacturing, based on a yield of 10% to 20%. The knee in the curve reflects that below about 100 units a year, a complete mask and wafer run would still be needed to produce the parts. The volume learning curve reflects the

reduced cost of design, testing, and handling on a per-chip basis as production increases. It also accounts for shifts in production techniques which occur with volume: while direct electron beam exposure is a cost-effective way to produce 1,000 units per year, it would be unlikely to compete with some form of projection lithography for chips produced in the millions of units per year.

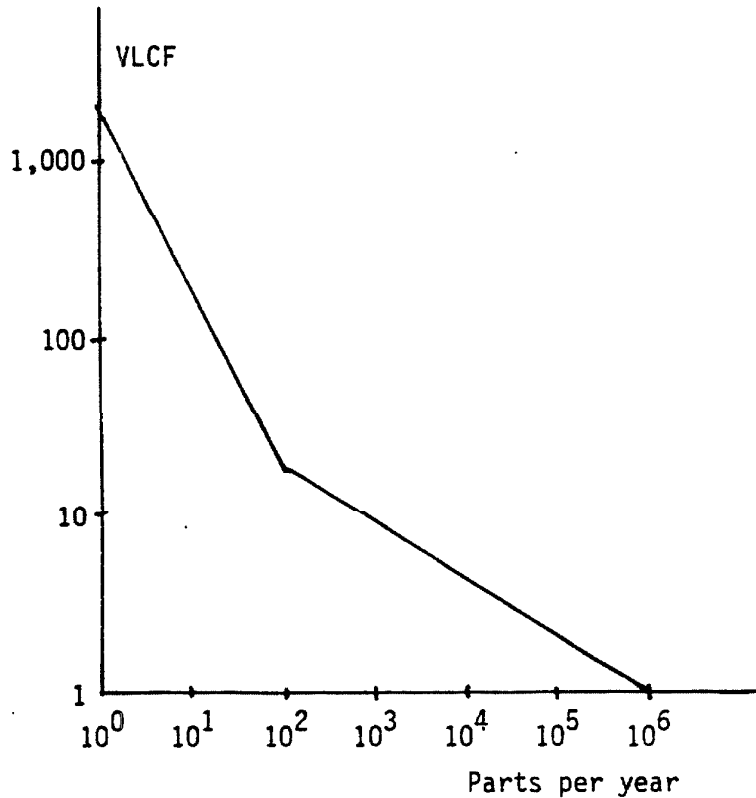


Figure 3-3. Volume Learning Curve Factor

The final factor in the chip cost equation provides a measure of the cost of chips which support the primary type of chip in the system. These chips will usually be more numerous but less expensive than the primary type of chip. These values represent estimates based on inspection of many board types, supported by some industry experience. Figure 3-4 summarizes the overhead factors used to account for support chips. These factors are multiplied by the cost calculated for the primary chip type to obtain the cost for chips in the subsystem.

Figure 3-4. Overhead Factors for Support Chips

<u>Primary Chip Type</u>	<u>Chip \$</u>	<u>Pins</u>	<u>Power</u>
Random Access Memory	1.1	1.5	2.0
CCD Memory *	1.1	1.5	2.0
Magnetic Bubble Memory	2.0	5.0	20.0
Content Addressable Memory *	1.1	1.5	2.0
Gate Array	1.0	1.1	1.1
Microprocessor	4.0	20.0	10.0

* Assumed to be same as RAM

3.2.2 Packaging Cost

The cost of packaging can be expressed as:

$$\text{COST}(\text{pkg}) = \text{NCHIPS} * \text{AVGPINS} * \text{OVERHEAD}(\text{pins}) \\ * (\text{ICPIN\$} * \text{BOARDPIN\$} + \text{SYSTEMPIN\$})$$

where

AVGPINS is the average number of pins per chip;

ICPIN\$ is the cost per pin of the integrated circuit package;

BOARDPIN\$ is the cost of the circuit board, on a per-pin basis;

SYSTEMPIN\$ is the cost of backplanes and cabinets, also expressed on a per-pin basis; and

OVERHEAD (pins) accounts for the pins of the supporting chips. The pin overhead factors are given by the second column of Figure 3-4.

The most inexpensive IC packages cost about \$0.25 for a 16-pin plastic package which can dissipate up to 400mW, for about \$0.015 per pin. Ceramic packages of the same size cost about \$0.80 and can dissipate up to 1 watt, thus costing about \$0.05

per pin. QIP packages of 64 pins cost about \$10, or \$0.16 per pin, for larger pinout requirements. IBM and Amdahl have developed packaging technologies for considerably more pins, but they consider cost data on these packages proprietary. In this study, pinout requirements of greater than 64 pins were estimated to cost \$0.30 per pin.

The circuit board which holds chips accounts for the largest part of the packaging cost. A typical printed circuit board might cost \$200 for board materials and processing, to which assembly and test labor (*) of \$400 must be added. If our hypothetical board sported 150 chips with an average of 20 pins each, this would imply a per-pin cost of \$0.20. Boards made by more expensive techniques such as wire-wrap or multilayer techniques were assumed to cost \$0.50 per pin.

If a backplane assembly costing \$1000 supported 20 such boards and fit into a cabinet also costing \$1000, this would add about \$0.03 per pin for system packaging.

(*) 16 hours at \$25 per hour, fully burdened labor cost.

3.2.3 Power

Power and cooling represent the final element of cost of these kinds of subsystems. Power cost can be expressed as:

$$\begin{aligned} \text{COST (power)} &= \text{NCHIPS} * \text{AVGWATTS} \\ &* \text{OVERHEAD (power)} * \text{WATTCOST} \end{aligned}$$

where WATTCOST is assumed to be \$5 per watt for passive cooling, \$10 per watt for forced air cooling, and \$20 per watt for liquid cooling. OVERHEAD accounts for the power consumed by support chips. The third column of Figure 3-4 gives the power overhead factors used. In the case of bubble memories, the support chips consume considerable power because they must drive the field coils of the bubble memory package.

3.3 NON-ELECTRONIC COMPONENTS

The largest capacity disk drives have remained remarkably steady in price over the last 15 to 20 years, while doubling in density about every three years. If this trend continues, large-capacity disk drives in 1985 will be priced at about \$25,000 to \$35,000 per spindle to end users and \$12,000 OEM, but have storage capacity of 2 billion bytes. With these prices and pricing mark-ups typical for industry, these drives will thus have a likely UMC of \$8,000. (*) Figure 3-5 shows trends in disk density.

Rack mountable 14-inch diameter drives lagged far behind the largest disks in density until the relatively recent introduction of the sealed-environment "Winchester" technology originally developed for the large drives. Starting in 1978, the density of these drives has increased dramatically [Elec78], to the point where the recording density (**) of the most advanced of these drives lags the largest disks by about a factor of two. With fewer platters, less mechanical loading on the positioning mechanisms, and electronics further down the learning curve,

(*) Average mark-ups on end-user products in the computer industry are about 3 to 1 over UMC; for OEM products the mark-up is about 50%.

(**) bits/square inch recorded on the media

these drives can be produced at a UMC of about \$3,000 to sell for \$5,000 to OEMs.

A similar movement is afoot in even smaller capacity drives [Durniak79], as manufacturers apply the sealed-environment technology to 8-inch drives. These drives are about another factor of two lower in bit density on the media, and use even simpler positioning mechanics than their 14-inch cousins. They are expected to supplant 8-inch soft disks. With OEM prices in the range of \$1,300 to \$2,000, these drives probably have a UMC of \$1,000.

It is only logical to assume that this process could be carried out again with 5-inch disks, with recording density another factor of two lower still. These units are expected to be produced with a UMC of \$300 to sell for under \$1,000.

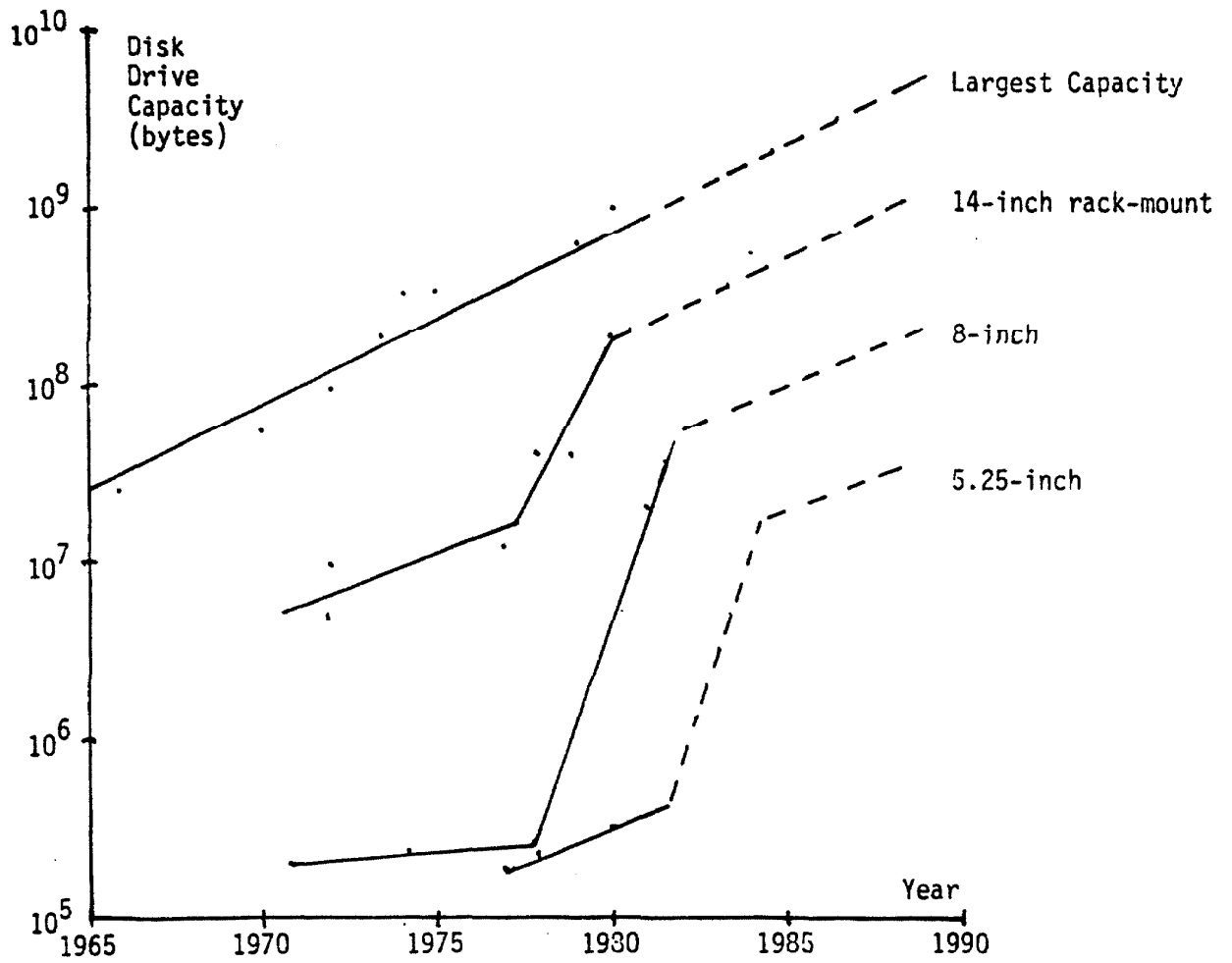


Figure 3-5. Disk Drive Capacity

3.4 SUBSYSTEMS

With the electronics cost model developed in previous sections, the cost of several interesting subsystems can be obtained. These subsystems will be used repeatedly in the following chapters. 1985 was selected as the target date for comparisons because it is far enough into the future to make for interesting and useful results, but close enough to the present to give some feeling for the reasonableness of the results. (*) Figure 3-6 summarizes the assumptions and results of the electronics cost model as applied to these subsystems. Several comments on that table are in order:

RAM. Although 1 megabit RAM chips should be available in 1985 according to the technology prognosticators, the maturity learning curve of Figure 3-6 suggests that they will still be too expensive, and that the more mature 256 Kbit chips will be enough cheaper to offset the saving in packaging and power. Thus, the technology of choice used the 256 Kbit parts 4 years into the

(*) Since this technology forecast was first written in late 1979, it appears that actual progress in silicon technology has fallen two to three years behind the pace predicted in the middle and late 1970's. Thus, although the text discusses 1985 as the target year for subsystems, 1987 to 1988 would be more realistic dates for the target costs shown. The text still refers to 1985, however, to remain consistent with the sources referenced as the basis of the estimates.

learning curve. The lowest power and packaging costs were assumed. Some of the subsystems were costed as though they contained some RAM; in these cases a price of \$1,500 per megabyte prevailed.

CCD. With the same underlying process, CCD memory can obtain about four times the number of bits per chip of comparable die size [Guidry78], due to both smaller basic cell size and less overhead circuitry on-chip. However, CCD's lag about two years behind RAMs on the learning curve. With other assumptions the same as for RAM, CCD results in one-third of the per-megabyte cost about for RAM. This result agrees with the forecasts of [Guidry78].

Bubble. Projections for bubble memories are not as boldly published as for their silicon competitors. Bubble memories can be packed to a bit spacing of about 4 bubble diameters [Hu78], which is roughly comparable to CCD's. However, with the major-loop/minor-loop organization commonly used, external control circuitry, and larger die sizes made possible by only a single critical mask alignment during processing, (*) bubble chips could have about 4 times the density of CCD memory. However, bubble memory chips will have very high support costs in terms of number

(*) versus 4 to 8 such steps for silicon chips

of support chips, their packaging requirements, and power consumption.

Microprocessors would contain over 100,000 transistors, and could be the equivalent of today's small mainframe in functionality and instruction processing power (about 0.5 million instructions per second (MIPS)).

Disk Controller. The mainframe version is assumed to contain 25,000 gates implemented with a gate array technology of about 6,400 gates per master-slice chip and 256 KBytes of memory. It would be delivered in quantities of 1,000 to 10,000 per year, utilize the most expensive IC and board packaging options, and be forced-air cooled. Its minicomputer-oriented cousin is assumed to have 10,000 gates and 128 KBytes of memory, utilize less expensive board level packaging, and be delivered in quantities of over 10,000 per year.

Mainframes. The projected 1-MIPS mainframe is assumed to have 150,000 gates and 1 MByte of RAM for control store, with expensive packaging and forced air cooling. The high-performance (16-MIPS) version contains 500,000 gates and would be liquid cooled. The high-performance version also would contain a 1 Mbyte CAM acting as a cache. It is interesting to note that both the 1-MIPS and 16-MIPS mainframes exhibit approximately the same cost per MIPS.

For comparison, the same model was applied to older technology mainframes. The "1975 Mainframe" represents the Amdahl 470V/6 as described in [Wu78]. In the 1975 mainframe, packaging and power represent 53% of the cost of the system, up from 43% for the 1965 model. However, with increasing levels of integration and rising chip cost, the 1985 mainframe puts only 23% of its cost into packaging and power, reversing the trend of prior years.

The same effect shows up even more dramatically in memory systems. A 1975 memory based on 1K RAM chips would put 77% of the system cost into packaging and power. A 1979 memory system based on 16K parts would still put 72% of the cost into boards, backplanes, and the like. However, with 64K chips, the balance swings the other way, with packaging and power accounting for only 28% of system costs; this declines to 22% with 256K RAM chips.

Figure 3-6. Subsystem Cost Calculations

<u>Subsystem Cost Component</u>	<u>1 MB RAM</u>	<u>1 MB CCD</u>	<u>1 MB Bubble</u>	<u>Micro- processor</u>
Number of Chips	32	8	2	1
Pins per Chip	20	20	20	40
Watts per Chip	0.2	0.2	0.2	0.5
Chip Overhead Factor	1.1	1.1	2	4
Power Overhead Factor	2	2	20	10
Mature Chip Cost	20	20	20	20
Maturity LCF	1.6	2.5	1.6	1.6
Volume LCF	1	1	1	5
Package Cost - IC	0.05	0.05	0.05	0.05
Board	0.20	0.20	0.20	0.20
System	0.03	0.03	0.03	0.03
Power Cost per Watt	5	5	5	5
<hr/> Subsystem Cost	<hr/> \$1500	<hr/> \$525	<hr/> \$300	<hr/> \$900

Figure 3-6 (continued)
 Subsystem Cost Calculations

<u>Subsystem Cost Component</u>	<u>Mainfr Disk Ctrlr</u>	<u>Mini Disk Ctrlr</u>	<u>1 MIPS Mainfr</u>	<u>16 MIPS Mainfr</u>
Number of Chips	4	2	25	80
Pins per Chip	150	150	150	200
Watts per Chip	2	1	2	5
Chip Overhead Factor	1	1	1	1
Pin Overhead Factor	1.1	1.1	1.1	1.1
Power Overhead Factor	1.1	1.1	1.1	1.1
Mature Chip Cost	20	20	20	20
Maturity LCF	1.6	1.6	1.6	3
Volume LCF	20	10	10	20
Package Cost - IC	0.30	0.30	0.30	0.30
Board	0.50	0.20	0.50	0.50
System	0.03	0.03	0.03	0.03
Power Cost per Watt	10	5	10	25
Extra RAM	256KB	128KB	1MB	1MB
<u>Subsystem Cost</u>	<u>\$3600</u>	<u>\$950</u>	<u>\$13.5K</u>	<u>\$125K</u>

Figure 3-6 (continued)
 Subsystem Cost Calculation

Subsystem Cost Component	Cache:	1975	1965
	1 MB CAM	4 MIPS CPU	1 MIPS CPU
Number of Chips	128	2K	4K
Pins per Chip	40	80	40
Watts per Chip	1	3.5	3.5
Chip Overhead Factor	1.1	1	1
Pin Overhead Factor	1.5	1.1	1.1
Power Overhead Factor	2	1.1	1.1
Mature Chip Cost	20	1	1
Maturity LCF	3	5	5
Volume LCF	5	20	20
Package Cost - IC	0.30	0.30	0.30
Board	0.50	0.50	0.50
System	0.03	0.03	0.03
Power Cost per Watt	10	10	10
<u>Subsystem Cost</u>	<u>\$48K</u>	<u>\$423K</u>	<u>\$700K</u>

Figure 3-6 (continued)
 Subsystem Cost Calculation

<u>Subsystem Cost Component</u>	<u>1980 1MB RAM</u>	<u>1978 1MB RAM</u>	<u>1975 1MB RAM</u>
Number of Chips	128	2K	8K
Pins per Chip	40	16	16
Watts per Chip	0.2	0.2	0.2
Chip Overhead Factor	1.1	1.1	1.1
Pin Overhead Factor	1.1	1.1	1.1
Power Overhead Factor	2	2	2
Mature Chip Cost	8.50	2.00	1.50
Maturity LCF	1	1.6	1.6
Volume LCF	1	1	1
Package Cost - IC	0.05	0.05	0.05
Board	0.20	0.20	0.20
System	0.03	0.03	0.03
Cost per Watt	5	5	5
<u>Subsystem Cost</u>	<u>\$4.9K</u>	<u>\$21.2K</u>	<u>\$93.1K</u>

CHAPTER 4 TRIAL ARCHITECTURES

4.1 INTRODUCTION

4.1.1 Architectures Considered

Since E. F. Codd first put forward the relational model for data base organization [Codd70], it has been widely recognized that the simple, tabular nature of the relational model makes it amenable to alternatives to the standard, von Neumann architecture. Many workers have proposed alternatives for the organization of a data processing system for data base processing, especially for relational data bases. (*)

Will these machine organizations be effective as vehicles for implementation of an advanced functionality data management system, as exemplified by the REL system prototype? Will a more conventional architecture be as effective, or will some architecture not envisioned in existing literature be the best solution?

(*) [Smith79] provides an excellent overview of work to date, including an extensive bibliography.

In an attempt to answer that question, a number of candidate architectures were selected and modeled using the program described in the previous chapter. These architectures fall into about four major classes:

(1). In a conventional, von-Neumann architecture, the data base is stored in pages on a secondary storage device. Pages of information are moved into the main memory, to be operated on by either a single processor or a shared-memory multiprocessor of conventional design. Architectures with both ordinary disk technology and some of the emerging "gap filler" technologies were considered.

The von-Neumann architecture remains an interesting candidate for a future "REL machine" for several reasons. Much of the time spent in processing a sentence in the REL prototype system today goes into transmission of pages between primary and secondary memory. The "gap filler" technologies could considerably reduce this time. Transmission of pages is already well optimized in the REL prototype [Greenfeld72], and further improvement is possible. Finally, a data base management system has many functions besides computing relational data base operators, and the merits or demerits of using a general purpose device for all of these functions should not be dismissed without analysis.

(2). A variety of proposals have been made for using some sort of associative logic with every "loop" of secondary memory. These architectures contemplate a serial secondary memory, either inertial (*) (as with disk or CCD) or non-inertial (bubble can be operated this way). The most advanced of these proposals contemplate using a small, fast content-addressable memory as a part of the logic per loop, and thus can be deemed "doubly associative" [Shaw79]. Several variants on this architecture were explored utilizing logic-per-head disk or an all-electronic analog with bubble or CCD memory serving as the secondary associative memory level.

(3). The parallel processing architectures proposed in the literature generally distribute work of fairly small "grain size," where each processing element works on either a character or even a bit at a time. It could be that an architecture could be extremely effective with a larger grain size for distribution of work. Consequently, one of the series of architectures locks at distributing work with the grain size of a record or block of records.

(4). A final set of cases considers implementations based on very large main memories.

(*) that is, the rotation or shifting cannot be easily stopped.

Each architecture represents a way to arrange logic and memory elements into a data base processing subsystem.

Another design issue lies in how many users each of these subsystems should serve: one, a few, or all of the users in the organization which owns the "REL machine"? This choice leads to three major sub-architectures for each architecture studied: "smart terminal," clustered, and central.

In a central architecture, there is one data base processing subsystem which is multiplexed among all of the users. Users are assumed to have "dumb" (*) terminals, which connect through a general-purpose computer used as a "front-end" to the "back-end" data base processing subsystem. All interversion communications (**) occur strictly within the confines of the single back-end resource.

(*) Here, we mean "dumb" in the sense of having none of the applications processing software. It must be expected, however, that future terminals will devote a considerable number of cycles to providing a "friendly" interface to the human user, through powerful screen formatting, graphics, and local editing. While such functions as editing account for a large fraction of the processing power consumed in the central mainframe in many contemporary systems, here we assume that such functions will be distributed to the terminal in all of the architectures studied, and are thus not a variable in this research.

(**) That is, communication between the data bases kept by one user and those of another user. The concepts of "interversion communications," "lateral," and "based" as used in this thesis are those of [Yu80]. In that work, "lateral" communications is also called the "channeling operator."

In clustered architectures, a small number of users share a data base processing engine. As in the central architectures, a general-purpose computer serves as a front end, managing a number of "dumb" terminals. It is presumed that terminals belonging to users in the same working group (or other organizational entity) will be on the same cluster processor; therefore based (*) communication can occur within the confines of the cluster.

(*). Basing refers to the situation where all of the concepts of one data base (the "based-upon" version) are made available to the other or "working" version. The working version can extend the based-upon version with new vocabulary or modify data in the based-upon version. These changes, however, are stored in the working version and are visible only within either the working version or other versions based in turn on that working version. The based-upon version itself is not altered.

By contrast, changes made in the based-upon version are seen both by the based-upon version and the working version. Thus, the communication is "closely coupled" in the sense that changes to the based-upon version are instantly reflected in the working version, and one way in the sense that changing the based-upon version also changes the working version, but changes to the working version do not change the based-upon version.

Refer to [Yu80] for more information about basing.

Lateral (*) communications, however, will travel between clusters. Thus each clustered architecture contains an intercluster network modeled on ETHERNET [Metcalfe75]. The cluster general purpose processor generally provides the network interface.

(*) In "lateral communication" between two versions, the users of the two versions agree beforehand that the owner of the "responding" version will provide certain information from his data base to the owner of the "originating" version. They also agree on the vocabulary by which the required information will be known to both of them.

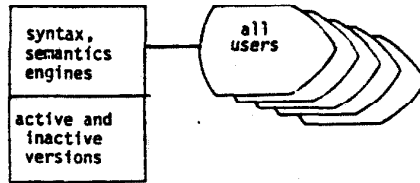
The owner of the originating version then puts a definition in his version for that vocabulary which refers to the responding version. The owner of the responding version similarly puts a definition of the vocabulary into his version, which translates the requester's terminology into terms which already exist in the responder's version. By appropriate choice of terms, the owner of the responding version can restrict the requestor's ability to interrogate the responding data base to only the agreed-upon information.

When the requestor's version parses a sentence which contains the vocabulary item which refers to the responding version, the system sends a request packet to the responding version. The responding version parses the packet using the definition placed in it just for that purpose, does the required processing, and returns the resulting information to the requesting version.

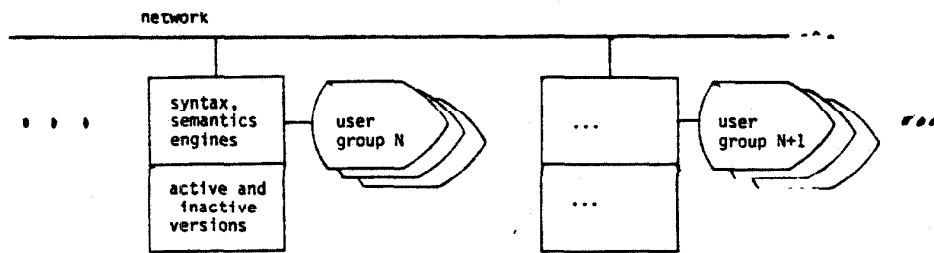
The reader should consult [Yu80] for more information interversion communication.

The "smart terminal" architectures presume that the data base processing engine is incorporated into the user's terminal, serving only that user. The data base engine must have sufficient storage capacity (possibly in several hierarchical levels) to contain an entire version. (*) Depending on the cost of the memory involved, the smart terminal may hold both the "active" version currently being processed and any inactive versions the user may have. If the memory is fairly expensive, it is assumed that only the active version resides in the smart terminal; an archival memory stores the inactive versions, which are loaded into the terminal over the network on demand. Because the smart terminal processes only a single version, any interversion communication uses the network.

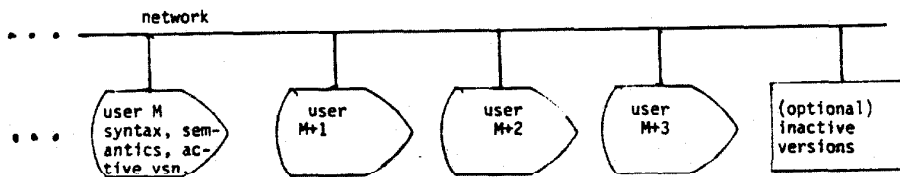
(*) In the REL system, the user organization's data are thought of as a number of data base/language pairs, each called a version. It is expected that each user would have several versions, representing perhaps different projects or areas of responsibility. This concept is similar in some respects to the conventional view of distributed data bases, but provides a different twist to the problem. [Thompson75] and [Yu80] discuss the REL view of versions at length and show how that view led to elegant solutions to the problems of distribution of control of an organization's data among many departments.



(a) Centralized



(b) Clustered



(c) Smart Terminal

Figure 4-1. User Sharing Alternatives

Cost functions representative of mainframe computers are used for the central architectures, whereas minicomputer or microcomputer cost functions are used for the cluster and smart terminal architectures. Thus, it makes for an interesting analysis to see if the lower cost of manufacturing due to volume economies of scale compensate for the generally higher overhead involved in the smart terminal and clustered architectures.

4.1.2 Assumed Load

The same load profile drove every architecture studied. The profile represents a mix of functions which does not necessarily correspond to that of any particular actual user organization. Instead, it was selected to exercise several kinds of functions which any data base processor would be expected to perform, ranging from fairly trivial requests to fairly complex operations.

In all cases, the model attempted to find configurations which would produce highly interactive response times. The criterion "highly interactive" must be expressed quantitatively, however, for the model to work. Thus the algorithm used required the weighted mean response time (*) to be between 1 and 15 seconds. Response times of under 1 second were taken to indicate "overkill": a configuration much too large for the user's purposes. Response times over 15 seconds seemed too long for an average response time of an interactive system.

The transaction mix used for all of the cases discussed in this chapter includes five transactions: logon, change version, simple query, simple sentence, and complex query. Code defining them can be found on page A-15.

The logon transaction represents a user beginning an interaction with the system. Processing this request requires examination of a small amount of data from a central system data base on all users to determine if the user is valid (proper password, authorized to use that terminal, etc.). In central and clustered architectures, this data resides in one of the versions on the configuration the user's terminal attaches to, and usually

(*) The sum of the relative frequency of occurrence times the mean response time across all transaction types.

can be retrieved quickly. In smart terminal architectures, the "authorized users" data base is presumed to reside on an archival store and must be accessed via the network. 1% of all transactions are logon requests.

The change version transaction moves a user from one version to another and constitutes 3% of the mix. In cases where the data base processing engine can store both active and inactive versions, this function can be handled by loading a few pages of control information. However, when the data base processing engine cannot hold both active and inactive versions, then the current version must be unloaded to the archive and the new version loaded. This obviously will put a significant load on the network.

The simple query transaction represents the parsing of an input sentence into a request which can be satisfied by 6 record retrieval operations. Assuming that the architecture in question has a way of mapping records onto storage in a deterministic manner (such as hashing), this usually translates into six I/O events to the data base. This transaction type is taken to be typical of trivial but frequent processing requests which ask for a few items of data at a time. This transaction type could represent a bank teller cashing a check or an inventory clerk recording picking of an item. It constitutes 32% of the mix.

For this transaction type, interversion communication is equivalent to obtaining the requested record from a remote file in the cluster and smart terminal architectures. This activity is modeled as the transmission of a request across the network, retrieval of the record at the remote configuration, and return of the contents of the record over the network to the configuration which originated the request. In central architectures, interversion communication has the cost of requiring changing versions (loading version control information pages).

The simple sentence transaction represents parsing an input sentence into a request to do one projection operation (*) and thus could be indicative of a user in a problem-solving situation. An example of this kind of user could be a bank officer examining a customer's account for a stopped check or an order clerk trying to find that lost purchase order number. The transaction constitutes 32% of the mix.

(*) Because of its reliance on only 1-ary and 2-ary relations (called "classes" and "relations" respectively), the REL system avoids most of the complexities of the relational algebra [Thompson75]. It can easily be shown that with the addition of dummy or placeholder entities ("current entities"), 1-ary and 2-ary relations can simulate any data structure possible with n-ary relations. A consequence of this conceptual simplification in REL is that the projection operator operating against a list or a computed condition is the basic data base manipulation operator of the system.

Interversion communication has different meaning for this type of transaction. Based operations represent activities against additional versions which must occur to satisfy the request. However, since information about the based version is stored in the requesting version's dictionary, no additional parsing is required. The request must be transmitted to the based-upon version. For central and cluster architectures, this transmission requires only loading the version control pages of the based-upon version. For the smart terminal architectures, however, the request must be transmitted over the net to another smart terminal configuration which stores and processes the based-upon version. That configuration then performs the indicated projection and sends the results back to the originating smart terminal to be merged with the results of doing the same projection on data stored in the requesting version. Modeling of based interversion communication in this fashion provides the same results even if the active version is based on several different versions.

Lateral interversion communication means that information stored in the receiving version's dictionary is used to interpret the request. The interpretation process is modeled by assuming that the receiving version must parse an input sentence contained by the request packet. The receiving version then does a projection and returns the results. In central architectures,

this result is available on the same media as both the requesting and receiving versions. In the cluster and smart terminal architectures, the receiving version executes on a different configuration, so the request packet and the resulting class must be transmitted on the network.

The complex query transaction represents use of the full facilities of an advanced data management system. Each transaction requires parsing an input sentence and performance of several "projection" like operations. Such an operation can either be a projection of extensional data (*) or expansion of the intentional data represented by a definition. A definition can expand into several projection operations and/or further definitions. Also, some of the definitions can involve lateral interversion communication. Further, since a lateral interversion communication involves parsing what is in effect a new sentence, even more work can result.

In the runs reported on in this chapter, a complex query does four projection operations in the absence of interversion communications. For each architecture, the level of based and lateral interversion communications was varied across a range of

(*) Data stored explicitly in the version. This data may also be augmented by data explicitly stored in a based version

values to test the sensitivity of the architecture to this added function.

The complex query transaction accounted for 32% of the mix.

4.1.3 Default Parameters

Each architecture model has many parameters. In order to insure that runs would be directly comparable, many of the parameters were either not varied or forced in the same way. Thus, the set of cases reported on here in no way exhausts the multi-dimensional space of possible cases.

Instead, a few parameters were investigated for each architecture in an attempt to gain some insight into the sensitivities of that architecture. The remaining parameters were given values which represent reasonable expected average values for "management information" applications.

The size of sets operated on by the relational data base operators of any architecture are given by several variables. CLASS_SIZE gives the number of items per class and was fixed at 500 for every run. #CLASSES gives the average number of classes per version, which was fixed at 50. The size of the 1-tuples in a class was fixed at 10 bytes.

To study the effect of data base size on the architectures, the parameter `RELATION_SIZE`, giving the average number of 2-tuples per relation attribute, was varied across a range of 500 to 500,000. The number of relations per version, `#RELATIONS`, was fixed at 100. All runs assume 20 bytes per 2-tuple.

The number of users was also forced across a range from 1 to 500 to study how different architectures behave in the face of widely varying size of user organizations. The degree of interversion communication was also forced across a range of values from none to a maximum of 50% of projections requiring access to an additional based version and 25% of the terms in an input sentence resulting in generating lateral queries to other versions.

For cluster and smart terminal architectures, an ethernet of at least 1 megabit/second bandwidth with length of 1 km transports packets of up to 4000 bits. In the cluster architectures, it was assumed that up to 6 terminals would share the cluster processor.

4.2 PAGING ARCHITECTURES

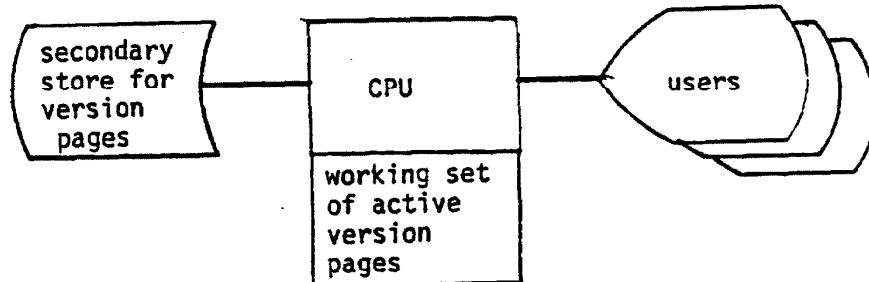


Figure 4-2. Paging Architecture

4.2.1 Disk

The paged architectures, particularly those employing disk, correspond most closely to the implementation of the REL prototype system. In these architectures, the secondary storage media holds the pages of versions which are actively being processed as well as the pages of inactive versions. A conventional von-Neumann computer does all of the work of parsing and applying semantic routines. The system can be thought of much as a timesharing system, which can only operate on information in the main memory of the computer, but whose performance critically depends on moving information between the main and secondary memory.

The design of the REL prototype system took optimization of page transmission as one of its primary goals. Norton Greenfeld's thesis [Greenfeld72] showed that appropriate choice of algorithms could achieve improvements of one or even two orders of magnitude above the behavior of the paging algorithms of conventional timesharing systems by using knowledge about how the data was physically stored on pages. These algorithms depend on maintaining data in lexical order on the pages, and sorting to lexical order when needed, in order to make most operations proceed in a time which is either linear with the size of the sets to be manipulated, or is at worst $n \cdot \log(n)$. (*)

It seemed natural to start with the REL prototype architecture of a central computer serving "dumb" terminals with disk the secondary store for several reasons. The architecture proved to be one of the simplest to model and provided a way to validate the modeling technique against measurements of an actual

(*) Any relational data base operation can be performed in linear time if the system maintains the data base "fully inverted," with the concomitant penalty in storage. "Inversion" refers to storing a duplicate copy of all of the tuples of the relation, where the duplicate copy is ordered for efficient retrieval by one of the secondary domain types. Thus, the relation can be accessed efficiently (which usually means in sorted order) by either the key or a non-key field without resorting it on the fly. Note, however, that if a tuple is updated, all of the duplicate copies of the relation (and there might be several) must also be updated, otherwise the data base will become inconsistent.

system. Also, since disk can be expected to remain the lowest cost per bit secondary store for the foreseeable future, configurations involving disk have good potential as trial architectures.

To validate the model, a set of 62 sentences was run against the NAVY data base, which gives some 80 attributes about some 500 naval and commercial ships, under the REL prototype system as a batch job at 3:00 a.m. With no other users on Caltech's IBM 3032, the system responded to each sentence in an average of 4.1 seconds. A run of the model with the same circumstances as input (no other users, 2.5 MIPS computer, 500 tuples/relation) gave a predicted response time of 3.84 seconds: within 7% of the measured value.

While disks have increased in density, as discussed in the last chapter, their access rates, as limited by the times to physically move electromechanical assemblies, have remained fairly constant. All of the disk configurations in this thesis use present-day parameters for access rates: average seek time of 30 milliseconds, rotational speed of 3600 RPM, and track size of 10,000 characters. It was also assumed that, as today, a disk controller could have not more than 8 drives attached. Disks were operated with a 2,000 byte page size - the same as in the REL prototype system and a recommendation reported in [Greenfeld72].

Three nodes sufficed to describe the centralized paging disk architecture: the CPU, disk controller(s), and disk drive(s). Each of the nodes serves all of the users.

Retrieval of a page represents the basic operation of this type of system; all activities are modeled in terms of page retrievals and additional CPU activity. Retrieval of a page involves execution of 2,000 CPU instructions in access method routines. The time taken by the peripheral was divided between the disk drive node and the disk controller node. The disk drive node provides only the seek time, modeling each drive as an independent server with its own queue.

The disk controller node provides the rotational latency, transfer time, and queueing delays associated with freeing up the controller to main memory channel. This arrangement may seem a little unnatural because rotational delay and transfer time are properties of the drive, and data streams through the controller with a delay of perhaps only microseconds. However, it permits the model to account for the fact that a disk controller transfers data for only one drive at a time, no matter how many are connected, because it is not buffered and has a bandwidth comparable to that of the drive itself. Where in the model the delay appears is really arbitrary, as long as it is properly accounted for somewhere.

For parsing the input sentence, the model assumes that the entire dictionary must be loaded into main memory: all of its pages are transmitted. The internal parsing time is given by a constant term for initializing the parser, a term proportional to the log of the number of rules in the dictionary which accounts for the binary search on rule strings done by the REL parser, and a term proportional to the number of words in the sentence. For the typical sentences modeled, the internal parsing algorithm executes about 36,500 machine instructions. (*)

The modeled parsing time compares closely with the figure of 0.1 second parsing time on an IBM 370/135 reported by [Thompson75]. The 370/135 has approximately 0.2 MIPS, so the parsing time implied by the reference is about 20,000 instructions. The small amount of time spent in parsing by the REL prototype system thus suggests that the attention of the designer of a REL machine would be better spent on the problem of relational data base semantic routine execution.

The model of the projection operation assumes that the data for both sets in the operation is stored on disk in the same

(*) Parsing is modeled in exactly this way for every architecture discussed in this thesis: loading of the entire dictionary into the general-purpose computing element of the configuration and execution of the same parsing algorithm.

lexical order. The projection operator thus brings in all of the pages of the relation and all of the pages of the class, producing an output set of pages with as many tuples as were in the smaller of the two input sets. This method of modeling the projection operator was used for all of the paging architectures, no matter what secondary storage medium was employed.

The cluster and smart terminal versions of the paging disk architecture view each configuration as a miniature version of the central architecture. The local configuration is modeled as the same three nodes: CPU, disk controller(s) and disk drives(s), with either one user (for the smart terminal case) or up to GROUP_SIZE users (for the cluster case). The remote configurations were modeled by aggregating all of the remote units into one set of three nodes, with more servers and more users. Thus these seemingly complex cases were modeled with only seven nodes.

Pages C-2, C-3, and C-4 show the computed cost functions for the central, cluster, and smart terminal variants of the paging disk architecture. None of the variants of the disk paging architecture could meet the 15-second response time objective for relations greater than 10,000 tuples.

As expected, the disk is the bottleneck. Since each disk drive is an independent server, any number of simultaneous users can be accommodated by adding drives. Contention for the disk proved not to be the problem, since the average queues for the disk drives and controllers were quite modest. Since any particular request can be served by only one drive, a series of requests will not be served any faster (except for reduced queue wait time) by more disks. Consequently, the problem lies in the low effective transfer rate of the disk drive itself. (*)

In order to meet the response time criterion with larger data bases, the effective transfer rate between the secondary store and the processor must somehow be increased. The remainder of this section considers other secondary store technologies which can be architected for faster transfer rates than can be achieved with disks. Other sections of this chapter consider architectures which attempt to increase the effective transfer rate between the storage system and the CPU by inserting some logic between the raw storage and the processor.

(*) The effective transfer rate of a disk drive is 33 pages, or around 66,000 bytes, per second. Only about 7% of the instantaneous transfer rate of a disk drive is thus actually utilizable because of the time taken to position the desired record under the read head. One might argue that the effective utilization of the bandwidth between the drive and the CPU is even lower, because most of the bytes on the page are irrelevant to the query.

The costs for the disk paging architectures are surprisingly low. For one or two users, the smart terminal variant proves least expensive. The cluster variant is best for between 5 and 20 users. The central architecture exhibits a striking economy of scale, however. This results from the relative underutilization of the CPU for few users. With few users, configurations require only one or two disks, so the CPU dominates system cost. As the number of users increases, more disks must be added to accommodate both their data storage and access requirements. The secondary store rises from 23% of the system cost for up to 20 users to 72% of the system cost for 500 users.

4.2.2 Magnetic Bubble Memory

Magnetic bubble domain technology has been promoted as an alternative to disk. Like disk, bubble memory retains data when powered off, making it attractive for use as a secondary storage medium. As the most prominently mentioned "gap filler" memory technology, bubbles seemed like the next candidate for investigation.

Compared to disks, bubble memories have virtually zero "seek" time, as the selection of the chip (or set of chips) to examine

utilizes simple multiplexers. Bubble memories have the equivalent of rotational latency, however. With major-loop/minor-loop organization, the expected time to get the first data bit out of a chip would be one-half the minor loop time, plus one-half the major loop time. The bubble memory then transfers out data bits, but at a much slower rate than a disk.

Bubble memories have an additional advantage over disks by being "non-inertial": the effect of rotating through the memory can be started and stopped with no loss of time. Because of this, it is possible to clock a number of bubble chips in synchronism. This makes it possible to interleave the bits of a record among several bubble chips, resulting in an effective bandwidth which can be arbitrarily large.

The paged bubble memory architectures differ from the disk-based architectures discussed in the preceding section in the characteristics of the secondary memory and how its GROW routine behaves. The GROW routine for bubble memories works by increasing the degree of interleaving needed to provide a data rate adequate to meet the response time objective. The degree of interleaving of the bubble memory doubled on every call to the grow routine, stopping only when the burst bit rate of the bubble memory exceeded that of an 8-way interleaved RAM memory 32 bits wide with 200 ns basic cycle time. (That limit was never reached.)

The bubble memory thus requires at least one chip for every degree of interleaving. After converging to the required degree of interleaving, the cost function routine would check to see if enough chips were provided to hold the requisite number of active and/or inactive versions. If not (as was invariably the case), the cost function calculates the proper number of chips.

Because of the expected higher cost of bubble memories, the smart terminal architecture includes an optical video disk memory on the network for archival storage of inactive versions. The OVD is assumed to have the same characteristics as a magnetic disk in terms of access time, but can store up to two orders of magnitude more bits per dollar. Each OVD has a communications microprocessor interfacing it to the network. This architecture thus has three levels of memory hierarchy.

Pages C-5, C-6, and C-7 show the cost function results for the central, cluster, and smart terminal variants on the paged bubble memory architecture. The results are most unsatisfactory, for several reasons. First, the bubble memory is more expensive per bit than disk. Second, and perhaps more disappointing, the overlapping scheme does not reduce rotational latency, which accounts for a major portion of the retrieval time. Thus, no degree of interleaving can increase the overall performance of the bubble memory by more than about a factor of two.

Consequently, the bubble memory becomes an inseparable bottleneck.

The tertiary memory scheme for the smart terminal case proved most unsatisfactory, as the time for downloading a version stretches into minutes even for small data bases. As a result, it proved impossible to keep the average time per transaction low. The times of the bubble memory on the terminal, the archival memory, and the network connecting them are roughly the same, and it would seem at first glance that overlapping them should be possible. Since the terminal's bubble memory serves only one user, there is nothing to overlap its operation. Because the network and the archival memory are shared, it is likely that successive requests to them will come from different terminals; thus they would not behave in an overlapped fashion from the point of view of one terminal.

4.2.3 CCD Memory

Charge-coupled devices (CCDs) make a very fast serial memory. Like bubble memory, CCD memory has no "seek" time. By comparison to bubble memory, however, CCD memory has a much faster "rotational" delay and transfer rate. CCD must probably be regarded as inertial, because stopping the clock for even a few

milliseconds will result in loss of data. However, since CCD's are clocked electrically, it is possible to use the same interleaving scheme discussed for bubbles.

CCD memory has the disadvantage of volatility: any CCD configuration must also include some non-volatile store at least for inactive versions. For the smart terminal architecture, this non-volatile memory was provided by an optical video disk archive, as with the bubble memory cases. The cluster and central cases presented here simply ignore the issue.

Except for the characteristics of the CCD chips themselves, the CCD paging cases were modeled in exactly the same manner as the bubble memory cases. The CCD memory is assumed to have a bit rate of 10 MBz, and a smaller page size than the bubble memory, as this turns out to maximize the throughput of a CCD configuration.

Pages C-8, C-9, and C-10 show the cost function results for the CCD paged central, cluster, and smart terminal architectures. With their vastly higher bit rates, CCD memories could handle most of the cases, although at a higher price than the disk configurations. Only the most demanding cases (over 100 users and relation size over 50,000) required a degree of interleaving greater than 1.

Because of the smaller optimum block size, the CCD cases reached CPU power limitations sooner, and thus could not handle the 200,000 and 500,000 tuple relation size cases. The smaller block size means more CPU overhead in I/O. Increasing the block size might permit the CCD architectures to handle these larger relation sizes within the prescribed response time, but would not significantly reduce the cost of these configurations, because the CCD memory dominates system cost.

While the CCD memory constitutes an overwhelming proportion of the cost, the CPU actually contributes the lion's share to response time, accounting for about 80% to delays. The CCD memory based architectures could not meet the response time objectives for the largest relation sizes because of CPU time rather than because of the paging memory.

4.2.4 Electron Beam Accessed Memory

Electron Beam Accessed Memory (EBAM) utilizes stored charge to represent information. A beam of electrons, deflected by either magnetic or electrostatic means, reads out the charge. Early pre-commercial computers used a form of EBAM memory, with a density of 16 to 64 bits per tube. History records that magnetic core memory quickly outstripped EBAM, and became the basis for all first generation commercial computers.

Interest in EBAM has revived in the last few years due to two major breakthroughs: silicon targets and two-level deflection systems [Smith78]. Proposed EBAMS use a target which acts like a FAMOS EPROM, where the beam addresses the bit needed, but with only one sense amplifier circuit for the entire target. The beam goes through a coarse-resolution deflector, which addresses perhaps a 128-by-128 array of fine deflectors.

Compared with other forms of memory, EBAM offers several intriguing advantages. Bit densities are limited by beam resolution, but the storage capacitors in the silicon target can apparently be as small as minimum geometry transistors (0.25 micron diameter). The EBAM target requires no patterning, and thus could be very inexpensive to fabricate. Information in an EBAM is relatively non-volatile: as with FAMOS EPROM, data could be stored reliably for 100 years before charge leakage became a serious problem.

EBAM has the disadvantage of requiring high voltage support circuitry. Progress in commercializing the new generation of EBAM has reportedly been slowed by difficulty in manufacturing the deflection mechanisms. Assuming that both of these objections could be overcome by reasonable cost, how would EBAM fare in the REL machine?

EBAM as a disk replacement has the interesting property of having neither seek time nor rotational latency. In this sense, EBAM acts more like slow RAM, providing a bit per tube approximately every 10 microseconds, with storage of 512 megabits per tube. (*) With this density and non-volatility, the three-level stores discussed for bubble and CCD are probably not necessary. Also, since EBAM timing is electronic, it should be possible to make interleaved EBAM systems with higher effective bit rates.

With the exception of the characteristics of the EBAM secondary store, the EBAM paging architectures were modeled exactly like their disk-based cousins. EBAMs were assumed to cost \$1,000 per tube, based solely on the forecast in [Smith78]. Each configuration includes an EBAM controller, which was assumed to cost exactly as much as a disk controller (for lack of any better way to estimate it).

Pages C-11, C-12, and C-13 present the cost results for the EBAM paged central, cluster, and smart terminal architectures. EBAM could be a very effective architecture, providing costs only slightly larger than disk for the smaller data base sizes, and throughput enough to provide real-time response to all but the very largest data bases through interleaving.

(*) based on the 1985 projection of [Smith78]

The principal of "equal frustration" gives one insight into why the EBAM architectures are effective. About 38% of the response time is due to the EBAM, with the rest contributed by the processor: a pretty fair balance.

[Smith78] suggests that EBAM could be used as a main memory with relatively slow cycle time but very large capacity. A later section in this chapter investigates the implications of an EBAM-main-memory architecture.

4.3 SERIAL ASSOCIATIVE ARCHITECTURES

4.3.1 Logic-Per-Head Disk

Many data base machine architectures have been proposed based on associating some logic with every "track" or "loop" of a serial secondary storage device. [Slotnick70] first explored the possibility of adding some comparison logic to the read/write logic of conventional disks and drums. This proposal led to a wide and growing literature of architectural proposals (and some experimental implementations).

The CASSM (Content Addressed Segment Sequential Memory) work of Lipoviski and Su represents one of the earliest proposals for harnessing logic-per-track systems to data base management. (*) CASSM could operate on the individual rows of a relation, marking all rows which matched a given pattern or had a given condition. These matches could then be output to a general-purpose computer for further processing. CASSM could be adapted to hierarchical data bases as well as relational data bases, but primarily served as a "filter" for selecting records which would then be processed further.

The RAP (Relational Array Processor) work of Ozkarahan et al. (**) took the next significant step in logic-per-track architectures, particularly those oriented to both disk and the relational model. RAP employs K comparators per track, and can compute K boolean conditions on the stored tuples in one revolution. Like CASSM, RAP is primarily concerned with marking rows of a relation that meet specified conditions, and outputting those rows to a general-purpose computer for further processing. Both RAP and CASSM deal with unordered data.

(*) CASSM first appears in the literature in [Healy72]. The literature includes several papers on the architecture of CASSM ([Copeland74a], [Lipoviski78] and [Su75]). The data manipulation language for CASSM is discussed in [Copeland74b], [Su77], and [Su78]. A performance analysis of CASSM appeared in [Chen76].

(**) See [Ozkarahan75] for an overview of RAP. [Ozkarahan76] more fully documents the concepts of the RAP project. [Schuster76] and [Ozkarahan77] discuss refinements of RAP.

The RARES (Rotating Associative RELational Store) system [Lin76] is another disk oriented design. In CASSM and RAP, tuples are arranged serially along the tracks of the disk. By contrast, RARES arranges tuples by placing the successive bytes of the tuple on adjacent tracks, so that the logic element scans a different tuple every character time. RARES uses a RAM called the response store to mark tuples which either meet the boolean condition to be computed or else are ready for output.

The CASSM, RAP, and RARES architectures are very effective at selecting the rows of a relation which meet a condition or set of conditions. Their effectiveness at computing the project and join operations, however, is limited by the number of conditions which can be evaluated at one time. CASSM searches for one condition at a time, and thus could be very slow at computing a projection of a large class against a relation. RAP computes projections and joins by using one of the K comparators per track for a tuple of the class, thus M/K passes through the relation must be made.

Noting the potentially high cost of projections and joins in RAP, [Shaw79] proposed using a content-addressable memory (CAM) as part of the logic per track. The CAMs are loaded with as many members of the smaller set as will fit. The CAM associated with each track is loaded with identical data. The serial memory then scans the larger relation, and any matches are recorded in the

CAM. It may take several cycles of loading the CAM and passing through the larger set on serial store to complete the computation.

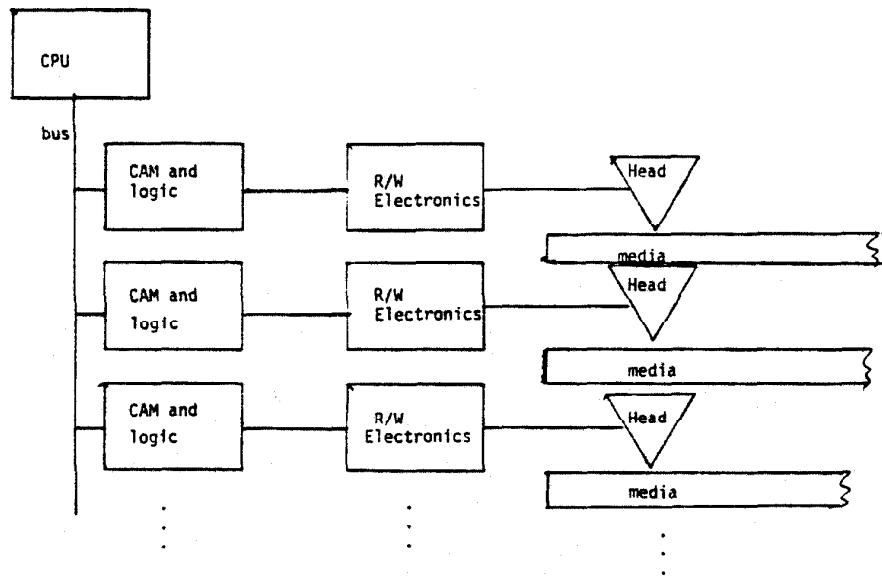


Figure 4-3. Logic Per Track Architecture

The serially associative architectures discussed in this thesis all have the form proposed by Shaw, employing a content-addressable semiconductor memory of high speed but limited capacity per track. [Locanthi77] gives a specification for such a CAM.

The disk modeled for the logic-per-head configuration assumed 20 recording surfaces (heads), the number typically configured on large-capacity disk drives today. The LPH disk drive is assumed to have a capacity of 1 billion characters, about one half of the density expected in the largest capacity drives, to account for the likelihood that LPH drives will develop further down the magnetic technology learning curve.

The HPT disk is modeled as three nodes: a "channel element," a "seek element," and a "rotating element." The rotating element models both the disk and the CAM. The time taken by this node represents the passage of data from the larger set passing under the heads and going to the CAM. The CAM is assumed to be fast enough to keep up with the data rate of the disk. The service rate of the rotating element node is expressed in terms of the burst data rate of the disk - about 1 million bytes/second per track.

The seek element accounts for head motion. It has a service rate commensurate with ordinary disks. The PAGE_SIZE variable is set at the number of bytes on a cylinder, which forces the model to calculate the appropriate number of seeks from the size of the relations.

The channel element accounts for reading matched tuples out of the CAMS and into the host computer. The channel element corresponds to the logic in the HPT disk drive which resolves contention between CAMS for use of the bus to the HPT controller, and then on to the host.

The HPT disk can be used as an ordinary disk with no reduction of throughput. The modeled cases assume that accesses to individual records and/or pages would be performed by access methods identical to those used for ordinary disks. It could be, however, that efficient organization of physical space on the disk for the projection operation might be sufficiently different from that appropriate for efficient single-record retrieval that any particular file would have to be organized one way or the other. If such a disparity in file organizations became extreme, it could greatly decrease the effectiveness of the HPT architectures if retrieval of single records required scanning entire files.

Projections are modeled by a fairly complex interaction of the seek, rotating, and channel elements. First, the number of passes through the relation required is computed by finding the cardinality of the smaller of the two sets and dividing it by the size of the CAM in tuples (any fraction is rounded up to the next integer). The CAMS must be loaded anew for each pass. The CAM

for each track of the disk will be loaded with an identical subset of the tuples of the smaller set. The CAM loading operation requires a seek, a rotational latency, a page transfer time, and an interrupt to the host CPU.

Next, the larger set is passed through the logic elements. The model computes the number of cylinders occupied by the larger set, and multiplies by the number of passes required to obtain the total number of cylinders of data which must pass under the logic elements. For each cylinder passed by the logic elements, a seek, rotational latency, and full track revolution time are required. At the end of the cylinder operation, the LPT drive generates an interrupt to the host CPU.

Matches are output to the central processor. If the projection in progress produces a partial result to a query, that partial result must be written back to the disk. This write-back operation occurs as ordinary page-by-page writes to the disk.

If either the seek element or the rotating element proves to be the worst delay node, the model first tries to decrease the number of passes required by increasing the capacity of the CAM. Because the CAM is a high cost per bit semiconductor logic/memory system, increasing the size of the CAM can increase the cost of the LPT drive fairly rapidly. The model will not try to expand

the CAM, however, if it is already larger than the smaller set, because this would not reduce the number of passes through the larger set (which is identically 1 for this size CAM). Instead, the number of LPT drives in the configuration will be increased.

If the channel element becomes the limiting node, the model will add another channel element, up to a total of 16 LPT controllers. If the CPU is the limiting node, it will be increased in processing power.

Pages C-17, C-18, and C-19 give results for the disk logic-per-head device applied to central, cluster, and smart terminal architectures. The smart terminal variant is least costly for one user, the cluster variant for 2 to 10 users, and the central variant for 20 or more users for all of the tested range of data base size. The LPH disk architectures appear able to cover the range of requirements with very low costs.

4.3.2 Bubble and CCD Logic-Per-Track

The concept of logic-per-track architecture applies to any serial storage medium, and most of the literature discusses these architectures without binding the implementation of the serial store to disk, bubble, or CCD memory. However, the actual choice

of the sequential medium has significant implications for the cost and performance of the resulting configuration.

Bubble and CCD memories offer several architectural alternatives which cannot be recognized in disk for logic-per-track systems. The design of the disk drive fixes the number of tracks. Further, because each drive in a string has its own motor and electromechanical access mechanism, it is generally not possible to synchronize disks closely enough to allow interleaved transfer of a block from several disks at once. However, with bubble and CCD memories, these architectural alternatives must be considered.

The bubble- and CCD-based logic-per-track architectures modeled here use the same fundamental organization as the disk logic-per-head model previously discussed. A relatively small but fast CAM stores the tuples of the smaller of the two sets which participate in a project operation, and the tuples of the larger set are passed by the CAM from the larger serial store.

However, the serial memory organization can be tailored by the architect to a much greater degree than is possible for disk. First, the degree of interleaving can be increased to provide a faster effective transfer rate. Secondly, the number of "tracks" can be set to any value, as opposed to the fixed number of

recording surfaces of the disk. Finally, the capacity CAM can be set, as discussed for the LPT disk architectures.

The bubble and CCD memories are thought of as being organized into a number of tracks, each with an associated logic element which includes a CAM. Much of the literature on bubble and CCD logic-per-track architectures assumes that the serial loop is large enough to contain $1/\#TRACKS$ of the data base, which leads to unreasonably long loop times in practice. The cases modeled here assume that each "track" is stored in a "major loop/minor loop" form, where major loops are addressed by conventional semiconductor logic multiplexers, and minor loops are implemented by the serial technology.

This organization permits the multiplexers to perform the same function as the access mechanism of the disk, effectively defining a "cylinder" as the same minor loop address in each "track." As with disk, single records within a track can be accessed in rotational latency plus transfer time. Similarly, the time to access the first record of a set for the projection operation is on the average one half the minor loop time, rather than one half the time to shift through $1/\#TRACKS$ of the entire data base. Unlike disk, the effective seek time is a matter of nanoseconds rather than milliseconds; therefore the bubble and CCD cases were modeled without the "seek element" used for the disk cases.

Both projections and single record retrievals are modeled for bubble and CCD logic-per-track architectures in much the same manner as for disk. For single record retrievals, the appropriate cylinder will be addressed electronically in negligible time, then rotational delay and transfer time are accumulated. For projection operations, the model computes the number of passes required through the smaller set based on the capacity of the CAM. The CAM is loaded by accessing the smaller relation, then the larger relation is passed under the CAM. The CAM of each track operates on the identical subset of the tuples of the smaller relation, but on disjoint subsets of the tuples of the larger relation.

The rules for growing bubble and CCD logic-per-track configurations differ from those for disk. First, the degree of interleaving is set so that the data rates of the serial memory and the CAM match. For CCD memories this means no interleaving because of the extremely fast data rate of the CCD chips. The number of tracks starts at one the capacity of the CAM at 16 tuples. If the logic-per-track memory becomes the worst delay node, the model will first attempt to decrease the number of passes through the data base by increasing the size of the CAM. When the CAM becomes large enough to hold the smaller set in its entirety, then further expansion of the CAM will do no good, so the model then increases the number of tracks.

Increasing either the number of tracks or the size of the CAM has the effect of increasing the number of comparisons done in parallel, which is proportional to the product of the size of the CAM times the number of tracks. Increasing either parameter will have the same impact on the cost of the configuration, as it increases the amount of CAM in the configuration but does not increase the amount of serial memory.

In many of the runs, a seemingly strange condition developed where the logic-per-track memory seemed to contribute little to response time while the CPU ran to saturation. This seemingly odd effect results from the generation of an interrupt at the end of each "cylinder" in the projection operation. To resolve this problem, the grow routines for these configurations must sometimes expand the number of tracks or the CAM capacity in order to reduce CPU utilization. Recognition of this problem and developing proper heuristics for the grow routines proved to be the major difficulty in modeling these configurations.

The heuristic which proved successful works as follows: when the model reports that the CPU is the worst delay node, the grow routine for the CPU node first computes what fraction of the CPU time is spent responding to interrupts for the projection process. If this fraction exceeds 40%, then the CPU node grow code simply branches to the logic-per-track grow code and does not expand the CPU at all.

Pages C-20, C-21, and C-22 show the cost function results for bubble-based logic-per-track memories for central, cluster, and smart terminal architectures respectively. Pages C-23, C-24, and C-25 show the cost function results for CCD based logic-per-track central, cluster, and smart terminal architectures respectively. The smart-terminal architectures for both bubble and CCD assume no archive. None of the CCD architectures address the need for the volatile contents of the CCD memory to be backed up by a non-volatile device.

In general, the bubble cases are all more expensive than their disk counterparts, and the CCD cases more expensive than the bubble counterparts. The increased expense reflects the fact that the cost of the serial memory dominates the cost of these architectures.

4.4 DISTRIBUTIVE FUNCTION ARCHITECTURES

Every database machine architecture can be characterized by a "grain size," or ratio of the size of the basic memory and logic element to the size of the entire system. The paging architectures exhibit a very large grain size, where the entire secondary memory and entire processing element are connected by a relatively small number of channels. The serial associative

architectures of the previous section exhibit a fairly small grain size, with relatively limited processing function (although a great deal of power in terms of bit rate) is matched by one track of serial store.

Other grain sizes can be envisioned. At the other extreme from the paging architectures, an architecture based on keeping the entire active version in a content-addressable semiconductor memory could be devised, with extremely high throughput and also extremely high cost for even modestly large data bases.

By contrast, this section contemplates architectures with fairly large grain size, that is, with a relatively large amount of logic, in the form of a microprocessor, associated with a fairly large amount of memory, in the form of a disk drive. A bus of some sort interconnects the microprocessors into a distributed function architecture which acts as a MIMD (*) multiprocessing environment.

Many systems of this character exist today, with widely varying degrees of cooperation between processors. The ETHERNET system represents a case where the processors cooperate only to the extent provided by applications-level programming

(*) Multiple Instruction Streams, Multiple Data streams.

[Metcalfe75]. More intimate cooperation, to the point of providing a distributed operating system which appears as a single entity to the user, can be found in the Tandem Guardian operating system and supporting hardware [Tandem76].

How would a multi-microprocessor system be architected as a REL machine? Obviously, computing relational database semantic operators should be the focus of the discussion, for we have already seen that parsing input sentences is not the problem.

Just as with the serial-associative architectures, assume that the distributed function architecture will be designed for relational data base semantic operators, and act as a "back end" to a general purpose computer which will handle parsing and I/O with the user. As with the architectures of the last section, a CHANNEL node connects the special-purpose relational data base "engine" with the general-purpose HOST computer.

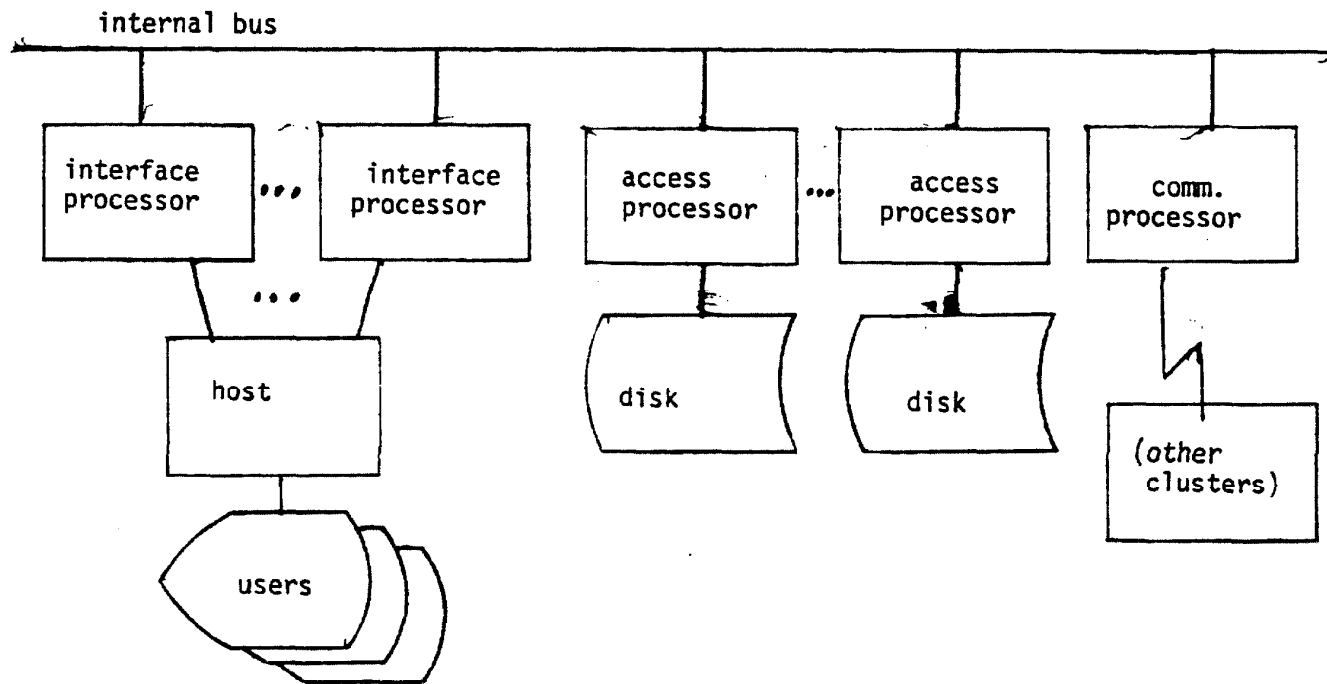


Figure 4-4. Distributed Function Architecture

The special purpose engine consists of two kinds of processors: INTERFACE PROCESSORS which connect to the host computer, and ACCESS PROCESSORS which provide the logic to

associate with one secondary memory (disk) unit. A BUS connects the processors in the configuration. To model the disk drive, a DISK CONTROLLER node accounts for rotational latency and transfer time, while a DISK ARM node accounts for seek time. (*)

To realize the parallelism inherent in this architecture, each access processor should store a portion of the data base, arranged such that a roughly equal portion of each relation resides on each access processor. This distribution of data has the same motivation as for the serial-associative architectures, where all of the CAMS should be kept busy for maximum throughput.

The required distribution of data to access processors could be achieved in a variety of ways. A hashing algorithm where the hash buckets correspond to access processors would be one of the simplest such algorithms. An algorithm based on key value ranges with a table of assignments of key ranges to processors would be equally suitable. In practice, any algorithm for mapping records onto processors which evenly distributes the tuples of a relation among access processors and is deterministic and easy to compute will suffice.

(*) This technique for modeling disk drives was discussed fully in Section 3.2.1.

Models of the distributed function architectures employ several nodes. The host is modeled as a uniprocessor, with mainframe cost functions for the central variant and minicomputer cost functions for the cluster and smart terminal variants. The host processor handles all I/O to the user's CRT and parses input sentences. Each retrieval or projection operation starts with a request generated by the host.

Requests from the host and responses from the back end flow over the channel. Each interface processor is assumed to have its own channel, with a bandwidth of one megabyte per second.

Interface and access processors are modeled as microcomputers. (*) The access processor is assumed to execute the same number of instructions for each kind of activity as were assumed for the disk paging architectures. (Remember that each access processor has only 1/Nth of the data for any relation, however). The models assume one disk drive per access processor.

(*) [DeWitt83] discusses a somewhat similar architecture known as the DIRECT system. DIRECT utilized minicomputers as the processing elements: a single VAX 11/750 for the interface processor and up to eight PDP 11/23's as access module processors. There are other differences in the allocation of function and the interprocessor interconnect schemes between DIRECT and the case discussed here.

The cluster and smart terminal variants on the architecture include a set of nodes for the local bus, access processors, and disks. In addition, the remote busses, access processors, and disks have a set of nodes to represent them in aggregate. Because of the functionality available in a microprocessor, the models assumed that both the local and remote back end engines would have a microprocessor on their respective busses dedicated to interfacing with the ETHERNET. Offloading the network interface task from the host processor was not possible in any of the other architectures because of the limited intelligence of the components.

The operation of the distributed architecture back end can best be illustrated by describing the flows for retrieving a record and performing a projection. A request for retrieval of a single record originates in the host processor and is communicated to the back end over the channel where it is fielded by an interface processor. The interface processor interprets the request and communicates it over the internal bus to the appropriate access processor. (*) The access processor then retrieves the record from disk in ordinary fashion, by reading in

(*) The record mapping algorithm must be deterministic because it would otherwise be necessary to communicate every single record request to every access processor, causing the system to do N times the amount of work actually required.

the entire block which contains the record. Software extracts the relevant bytes of the logical record requested, which is forwarded back to the interface processor via the bus and thence to the host via the channel.

Because the access processor extracts only the relevant bytes from the larger physical block on secondary store, the architecture utilizes considerably less bandwidth on the internal bus than is required between the processor and secondary store. By contrast, some other architectures result in passing considerable data, much of which is not relevant, to the processing elements.

The projection operation takes advantage of this property to an even greater degree. A projection operation similarly originates in the host and is communicated over the channel to an interface processor. The interface processor broadcasts the projection request over the bus to all of the access processors simultaneously.

Each access processor does the projection operation on $1/N$ th of the data. As with the paging architectures, assume that relations (in this case, portions of relations) are stored on disk in key sequence. If the projection is on the primary domain (key field) of the relation, then the data mapping algorithm

guarantees that tuples of the class and of the relation which could have matches will already be mapped onto the same processors.

If the projection operation is on a non-key domain, then tuples of the class will in general be in different access processors from the tuples of the relation which they will match. The relation must be resorted on the requested domain and redistributed via the bus under the mapping algorithm before the projection operation can be performed as for the previous case. Alternately, the tuples of the relation could have been stored in inverted form. Note inversion would make it possible to compute any desired projection without sorting; however, it also implies that a tuple and its inverted copy will generally map onto different access processors.

To complete a projection operation, the access processors could output their partial results via the bus to the interface processor, which would merge them and forward them to the host. Alternately, the access processors could store their partial results back on their disks for use in a subsequent step of processing a multi-projection sentence. The latter possibility could greatly reduce the requirements for bus bandwidth.

The models of distributed architectures start out with a single 0.5 MIPS host computer, which grows by 0.5 MIPS increments as needed. Each configuration starts with a single channel and interface processor. If either the channel or the interface processor must be expanded, the model adds another interface processor - channel pair.

Configurations start with two access processors and disks. If the access processor node (*) becomes the worst delay node, the model increases the number of access processors and associated disks by 10% rounded up to the nearest integer. The model permits the number of access processors to grow without limit; however, increasing the number of processors also increases the modeled time for a processor to acquire the bus and begin to transmit a packet.

When a configuration has converged to the response time criterion, the cost function for the distributed architecture picks the appropriate capacity disk. The program first tries to see if the entire data base will fit on 5-inch disks of 20 megabyte capacity at one drive per access processor. Next, the program checks to see if 8-inch drives of 100 megabyte capacity

(*) Remember that this QUEUETYPE=1 node models all of the access processors.

will be adequate; next come 14-inch drives of 500 megabytes; finally 2 gigabyte large-capacity drives are assumed. If the largest drives cannot accommodate the data base at one drive per access processor, then the program increases the number of access processors and 2 gigabyte drives until the configuration can accommodate the data base. After thus adjusting the cost of the disk and also perhaps the number of access processors, the program computes the cost of the configuration.

Configurations start out with a bus one byte wide with a cycle time of 150 nanoseconds. If the bus becomes the worst delay node, the bandwidth of the bus grows by increasing the number of data wires, up to a limit of 16 bytes wide. Increasing the bus width also increases the cost of each processor because it requires more bus drivers.

Pages C-14, C-15, and C-16 give cost function results for the disk distributed architectures in central, cluster, and smart terminal variants. Just as the paging architectures could be operated with other forms of secondary memory, the distributed architecture could also be defined with bubble, CCD, or EBAM as the secondary store. However, the results from the paging architectures suggested that with the lowest cost per hit, the disk-based cases would be the most interesting candidates for study.

In general, the smart terminal variant on the architecture proved better than the cluster or central variants only for one user. The cluster approach seems better for only two to five users; above that point the central architecture proves most cost effective. This is to say that one central configuration is always superior where communication with the terminals is well optimized.

4.5 ONE-LEVEL STORE ARCHITECTURES

4.5.1 RAM

The architectures considered to this point employ either two or three hierarchical levels of memory and logic for processing active versions. The design complexity of all of those architectures resulted from the need to manage the flow of information between levels of the memory/logic hierarchy.

Some of the experimental natural language systems treat memory as if only a single hierarchy existed. The addressing mechanisms of the hardware and paging mechanisms of timesharing operating systems encourage this view. Unfortunately, the mapping of data onto the secondary store may not be favorable,

which partially accounts for the slow performance of many of these systems for even modest sized data bases.

The MULTICS operating system represented one of the first attempts to provide the appearance of a one-level store. A MULTICS program issues address references to segments which may be resident either in main memory or on the paging device (which is typically disk), and the operating system resolves references to segments which are not in main memory by suspending the requestor until the segment is brought in from the disk. MULTICS treats a disk file as a special case of a segment.

The IBM System/38 [IBM78] represents the next step in one-level store architectures. System/38 programs issue 48-bit addresses, and do not know whether the requested byte was brought up from cache, main store, or disk.

From a performance standpoint, there would be no difference between the model of a disk/RAM one-level store and the disk paging architectures discussed early in this chapter, but only if both adopted the same paging algorithms. As discussed by [Greenfeld72], however, paging algorithms typically employed in such systems do not employ a strategy for locking crucial pages in real memory, and thus result in more paging than the optimized algorithms assumed for the paging architecture cases. Thus, the

reader is referred to Section 4.2 for a discussion of the best case for architectures where a one-level store is simulated with a fast main memory and a slower secondary store.

The simplest true one-level store is a RAM large enough to contain an entire active version. The primary advantage of such an architecture would be to simplify the programming of an advanced data management system by eliminating the need to choose data structures for efficient use of the secondary store. The simple list or ring structures which characterize some of the early and/or experimental natural language systems could be used in this kind of system with impunity from thrashing.

The model used here of an all-RAM one level store assumes that the RAM must store only active versions. Inactive versions are brought in from disk on demand. All of the flows were modeled as for the disk paging architecture, with two exceptions. The "change version" request means that the entire current active version must be copied out to disk and the new version read in. In all other requests, the software overhead involved in dealing with peripherals has been eliminated.

Pages C-29, C-30, and C-31 present the results for all RAM single-level store systems for central, cluster, and smart-terminal architectures. As the cost functions show, as nice an

idea as the all-RAM single level store may be for the software implementer, the cost will be prohibitive compared to the more complicated approaches discussed previously.

4.5.2 EBAM

EBAM was the original main memory of the earliest computers. [Smith78] has proposed that EBAM could make a comeback as a main memory because it is random-access in nature. Many EBAM tubes can be interleaved to create a memory with respectably high throughput, and if combined with a semiconductor cache memory, EBAM could make a very high capacity main memory. What, though, are the consequences of such a memory?

For one thing, the high degree of interleaving and long latency time compared to the speed of the processor (10 to 100 instruction times or more) mean that the EBAM would be best used with a wide cache wordsize and heavily favor algorithms which access data linearly. Fortunately, the projection operation has these characteristics.

With projection viewed as a linear scan through ordered sets, the flow of information between the EBAM and the cache would be just the same as in the EBAM paging case of flow between the EBAM

and main memory. The major difference would be that the cache hardware would manage transfer of pages between the EBAM and semiconductor memory without generating interrupts for the CPU or requiring a complex access method. Consequently, the performance of the EBAM single-level store architecture will closely resemble that of the paging architecture, but with lower CPU utilization.

Pages C-26, C-27, and C-28 show the cost function results of the EBAM single-level store architectures for central, cluster, and smart terminal variants. These runs were made by simple modification of the programs for EBAM paging architectures, where the CPU overhead per page was reduced from the 2,000 instructions appropriate to an access method execution to 100 instructions which would be more characteristic of setting internal memory mapping registers.

In all cases, the EBAM single-level memory cases are equal to or less expensive than their paging counterparts due to lower CPU costs. EBAM costs are identical.

CHAPTER 5 DISCUSSION OF THE RESULTS

5.1 INTRODUCTION

Before turning to the results of the study, it is worth considering what the "best architecture" means. Ideally, one architecture would be superior to all others for every combination of forcing parameters (*) conceivable. In the terms of this study, "superior" means lowest cost for the fixed level of response time.

Unfortunately, the data generated by this inquiry do not lead to such a clear-cut result. Instead, we find, not unexpectedly, that different architectures are optimal for different regions in the multi-dimensional space of organizational requirements. However, several general conclusions can be drawn: these are discussed in the ensuing sections.

Since there is no single optimal architecture for all cases, the definition of "best architecture" becomes perforce more difficult. The best architecture, under a weaker definition, would be optimum or reasonably close to optimum for more of the space of user requirements than any other architecture. It is

(*) see Figure 2-5.

this weaker criteria that will be used in arguing the conclusions of this chapter.

5.2 PARSING IS NOT THE PROBLEM

Almost all of the literature on the problem of advanced data management systems anticipates that the implementation of semantic operators will be more important than implementation of syntactic analysis (parsing). (*) The results of all thirty cases presented herein generally confirm this notion for all of the architectures studied. However, there are some important caveats to over-generalizing this conclusion.

Most of the literature which concludes that parsing is not the main problem considers the richer linguistic aspects of advanced data management systems. Another way of saying this is that it is typically assumed that the simple sentence or complex query transactions dominate the workload. Indeed, this is the case in the present study. As the sample result on Page B-6 shows, parsing accounted for 12% or less of the response time for the simple sentence transaction and 4% or less of the response time for the complex query transaction. Further, this relative contribution drops dramatically with increased typical relation size. (**)

(*) See, for example, [Smith79] or [Thompson75].

(**) The output shown in Appendix B assumes 5,000 tuples per relation: a relatively modest value.

However, the simple query transaction is a glaring exception, in which parsing typically accounted for 75% (or more) of the overall response time. This result must be expected to follow from the premise that just because the semantic operators involved are simpler, the task of analyzing the input sentence to determine which semantic operators to apply to which parts of the data base is no easier.

The foregoing observation has several implications for advanced data management systems. First, the promoters of such systems must recognize that they are not likely to perform as well as third generation (*) systems on simple, single row inquiry and update operations because of the overhead of parsing. Advanced systems offer a high degree of personnel productivity for such operations, however, because each input sentence is equivalent to a simple, but nonetheless exacting, program of perhaps 10 to 100 lines in an algorithmic language such as COBOL.

(*) See Figure 1-1.

A more severe problem lies in that data must initially be entered into any system, including a fourth generation system, typically through a large series of simple individual update transactions. This initial data load (*) thus may have a time penalty of about a factor of four when compared to present-day systems.

The solution to both problems lies in providing mechanisms by which the parsing effort can be amortized over a large number of simple cases. The results of the parsing operation must be saved in some fashion and executed repeatedly with different values instantiated for the variables.

Historically, this problem has been solved in one of two ways: by compilation or by an special facility. Compilation has the advantage that the same language which is customarily used in an interactive mode can be used to specify the operations, but the disadvantage of greater overhead associated with each parsing operation. (**)

(*) which may be part of a periodic (daily, weekly, or monthly) operationing routine

(**) See [Astrahan76] for a discussion of this approach in IBM's System/R prototype.

The REL Bulk Data Entry facility [reference??] exemplifies the second approach. The approach achieves similar efficiencies in that the data descriptors need be parsed only once, and the resulting semantic graph can be executed for each input record. In this approach, a sentence is permitted to contain variable words, and a list of constants can be input from a separate file. The sentence is parsed with the variable words only once, and the values in the constant list instantiated each time the resulting semantic graph was executed, until the constant list is exhausted.

Another alternative to either compilation or to a special facility would be to provide a cacheing mechanism for semantic graphs. The cache would record input sentences and resulting semantic graphs, and use the cached semantic graph whenever the same user input the same sentence. This approach would be particularly effective in combination with the variable list instantiation technique discussed above.

A sentence cache would be effective assuming that the same user often input the same sentence, as would be the case with variable list instantiation in an initial data entry situation. If the user were inputting a series of different sentences, the personnel productivity argument advanced above would tend to indicate that the users should not be concerned about the larger

parsing time of an advanced data management system compared to a less functionally rich system. The sentence cache has the problem that any change in database structures could invalidate some, none, or all of the semantics graphs which are cached. (*) The question of the proper policies for "spoiling" the cache is an interesting trade-off, which will not be considered here.

Most of the foregoing considerations for simple transactions apply generically to all of the architectures discussed, in that all were modeled with very similar assumptions for the mechanization of the parsing process. One opportunity for difference lies in the opportunity to process a number of simple transactions in parallel. As the execution of simple transactions is likely to be compute bound, this will favor architectures which can make maximum utilization of secondary storage access rates. The paging and serial-associative architectures (**) suffer in this regard because it is relatively expensive to configure few (one or two) secondary storage units per controller. The distributed function architecture (***) seems particularly well suited to this requirement, however.

(*) [Astrahan76] discusses similar problems for compiled transactions.

(**) see Sections 4.2 and 4.3 respectively.

(***) see Section 4.4.

5.3 USER SHARING ALTERNATIVES

Chapter 4 raised the question of how many users should a configuration serve: one, a few, or all -- corresponding to the smart terminal, clustered, and centralized variants on each major architecture. Figure 5-1 shows typical results for this comparison. A major finding of this study is that the relationship between the three variants remains much the same, despite major differences in the underlying basic architecture.

For a relatively large number of active users, a centralized approach is least costly. The clustered approach proved better than the smart terminal approach even for very small user populations. In general, more demanding organizational requirements (*) lead to the centralized approach becoming less costly than the others for a smaller number of users.

The major cost disadvantage of the smart terminal approach stems from the relatively low utilization of computational and storage resources. Since the cases were defined to have at worst an average response time of 15 seconds, and an arrival rate of one request per user every 60 seconds was assumed, the highest utilization possible for these cases is 25%. This utilization

(*) typical relation size, rate of lateral and based operations, etc.

disadvantage, even when weighed against manufacturing economies of scale, still results in the conclusion that the heavy resource demands of advanced data management systems will be most economically met by shared configurations for the foreseeable future.

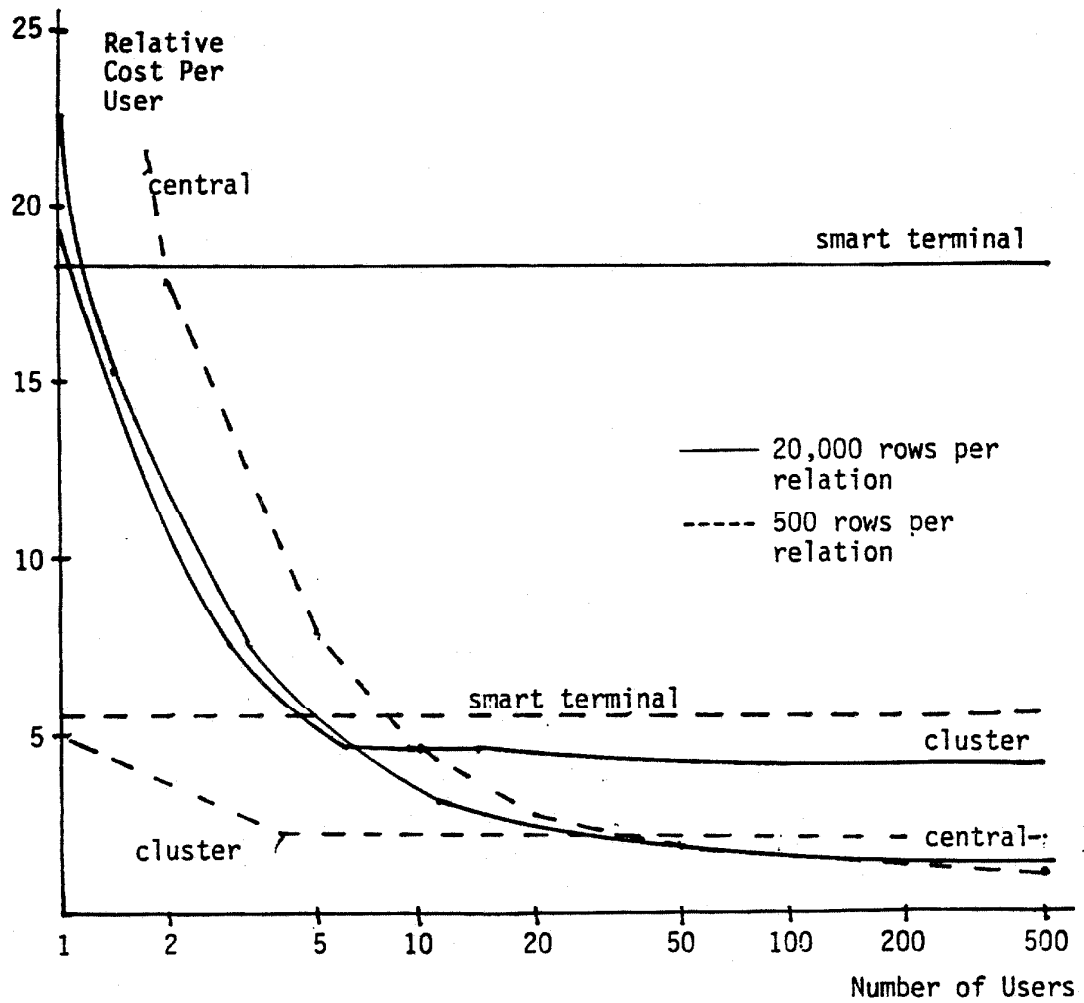


Figure 5-1. Relative Cost of Sharing Alternatives

Another problem with the smart terminal and clustered variants is the network bandwidth required when a large number of users share the network, and based and lateral operations on large classes or relations are common. Figure 5-2 shows the bandwidth requirements for a communities of 100 and 500 users

under both the smart terminal and clustered regemes as a function of the relation size.

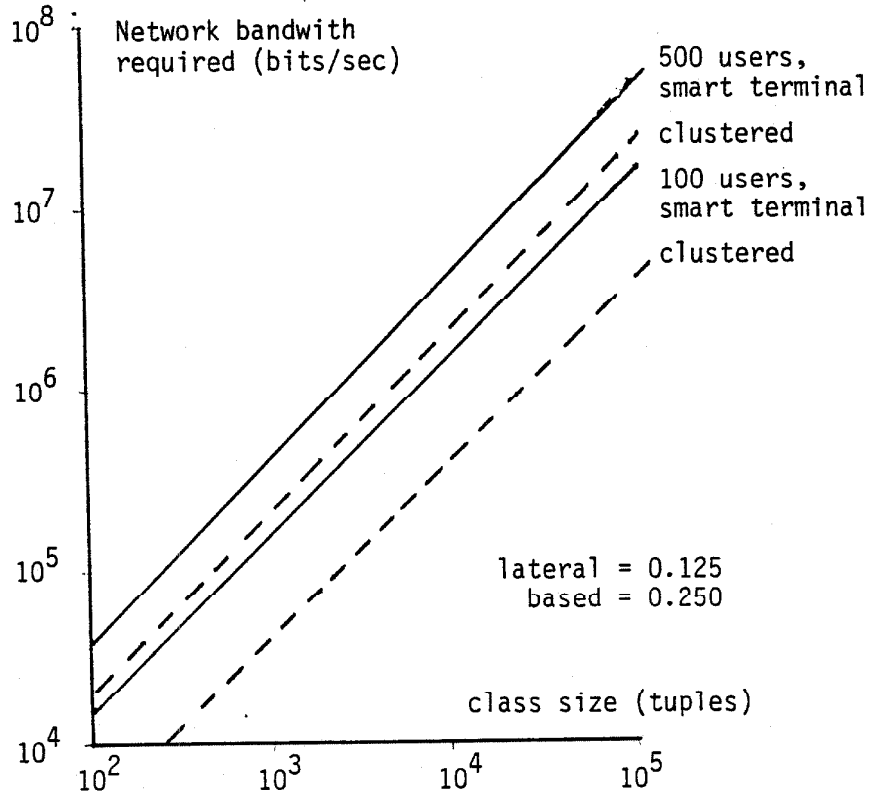


Figure 5-2. Network Bandwidth Requirements

5.4 MEMORY TECHNOLOGY

As [Greenfield72] observes, the time to transmit pages between levels of storage hierarchy represents a major contributor to response time. In the paged architectures, it is usually the dominant factor in response time. One of the objectives of this study was to quantify the impact of alternative memory technologies on the performance and cost of advanced data management systems.

Based on the cost model of Chapter 3, Figure 5-3 summarizes the per bit cost of the memory technologies considered.

Figure 5-3. Memory Technology Cost Forecast

<u>Technology</u>	<u>Capacity (bits)</u>	<u>Subsystem Cost (UMC)</u>	<u>Cost (milli- cents/bit)</u>
Largest disks	8.0 E9	\$8,000	0.10
14" OEM disks	1.6 E9	\$3,000	0.19
8" OEM disks	6.4 E8	\$1,000	0.16
5.25" OEM disks	1.6 E8	\$ 300	0.19
EBAM	5.2 E8	\$1,000	0.20
Bubble	8.0 E6	\$ 300	3.75
CCD	8.0 E6	\$ 525	6.56
RAM	8.0 E6	\$1,500	18.75

Figures 5-4 and 5-5 illustrate the impact of the various memory technologies on performance. In Figure 5-4, the y-intercept represents the latency of the device, including the CPU time to service the page request; the slope represents the transfer rate. Disk has a high latency and high transfer rate. Bubble has a more modest latency, but a relatively low transfer rate: thus bubble memory would seem to be optimized by a smaller page size. EBAM has essentially no latency other than software overhead, but a modest transfer rate. CCD is so fast that the latency is minimal and the transfer rate more than comparable to disk. The transfer rates of bubble, CCD, and EBAM can be improved by interleaving.

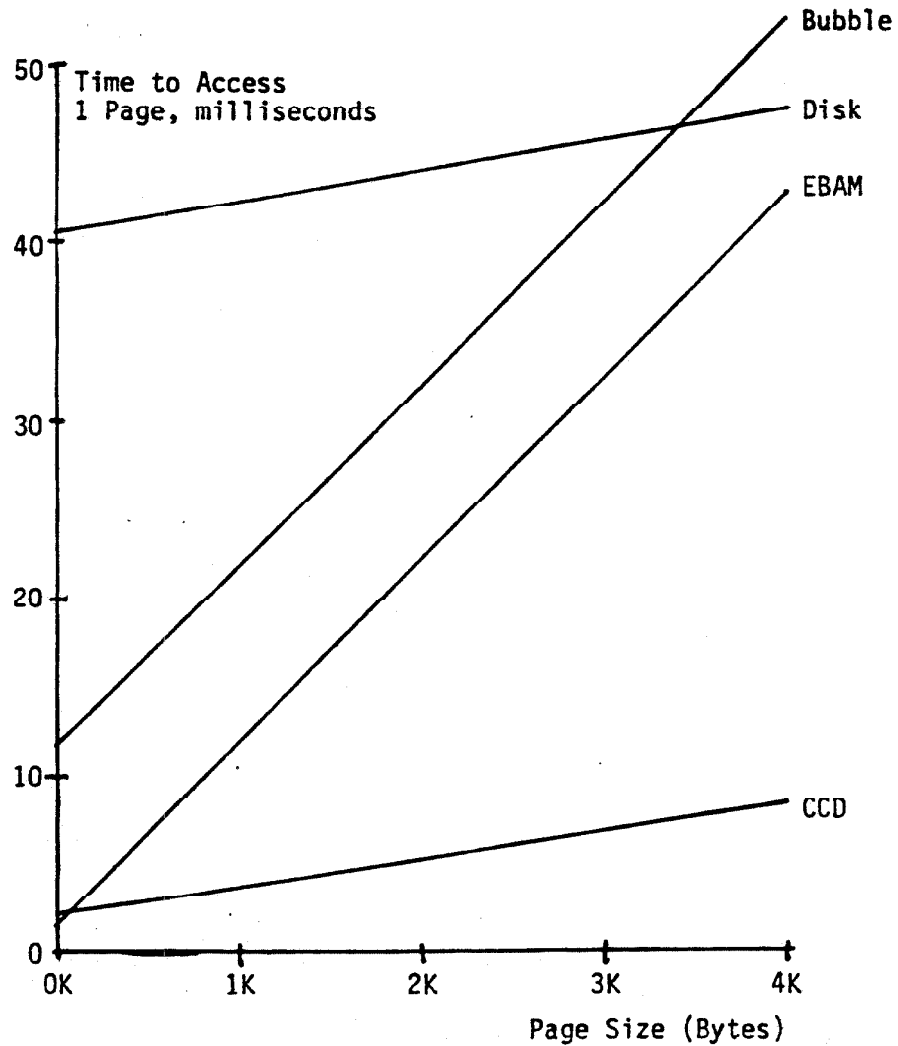


Figure 5-4. Page Transfer Time

Figure 5-5 presents a different interpretation which is more useful in making page size trade-offs. The tendency of bubble and CCD memory to be optimized at smaller page size is offset by the increased CPU time taken by transmitting more pages. Given a

fairly large page size (2 Kbytes to 4 Kbytes), disk, EBAM, and bubble memories result in comparable performance. However, as can be seen from Figure 5-3, the cost of bubble memory is unacceptably high. Similarly, although CCD offers a large performance advantage over disk, it does not compensate for the cost disadvantage.

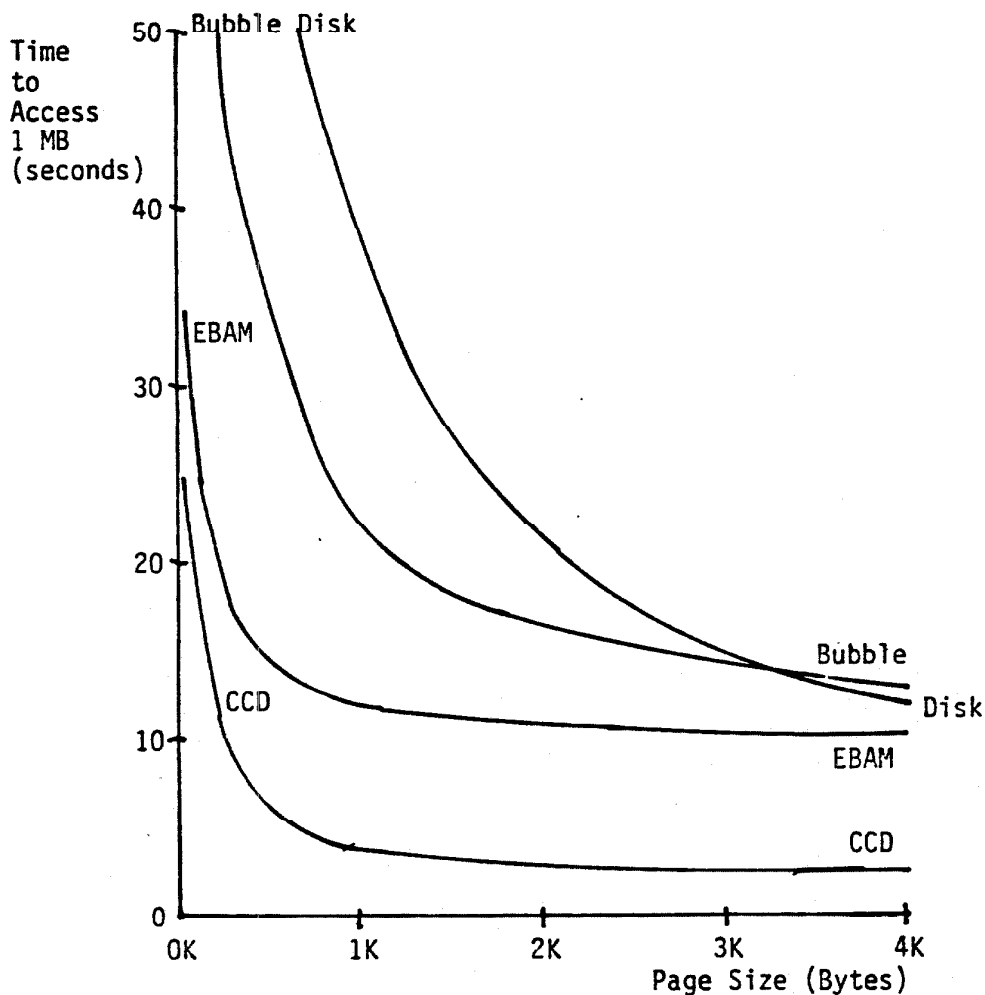


Figure 5-5. Time to Transfer a Megabyte

Figure 5-6 shows the cost functions for the various paging architectures as a function of the number of users and the memory technology.

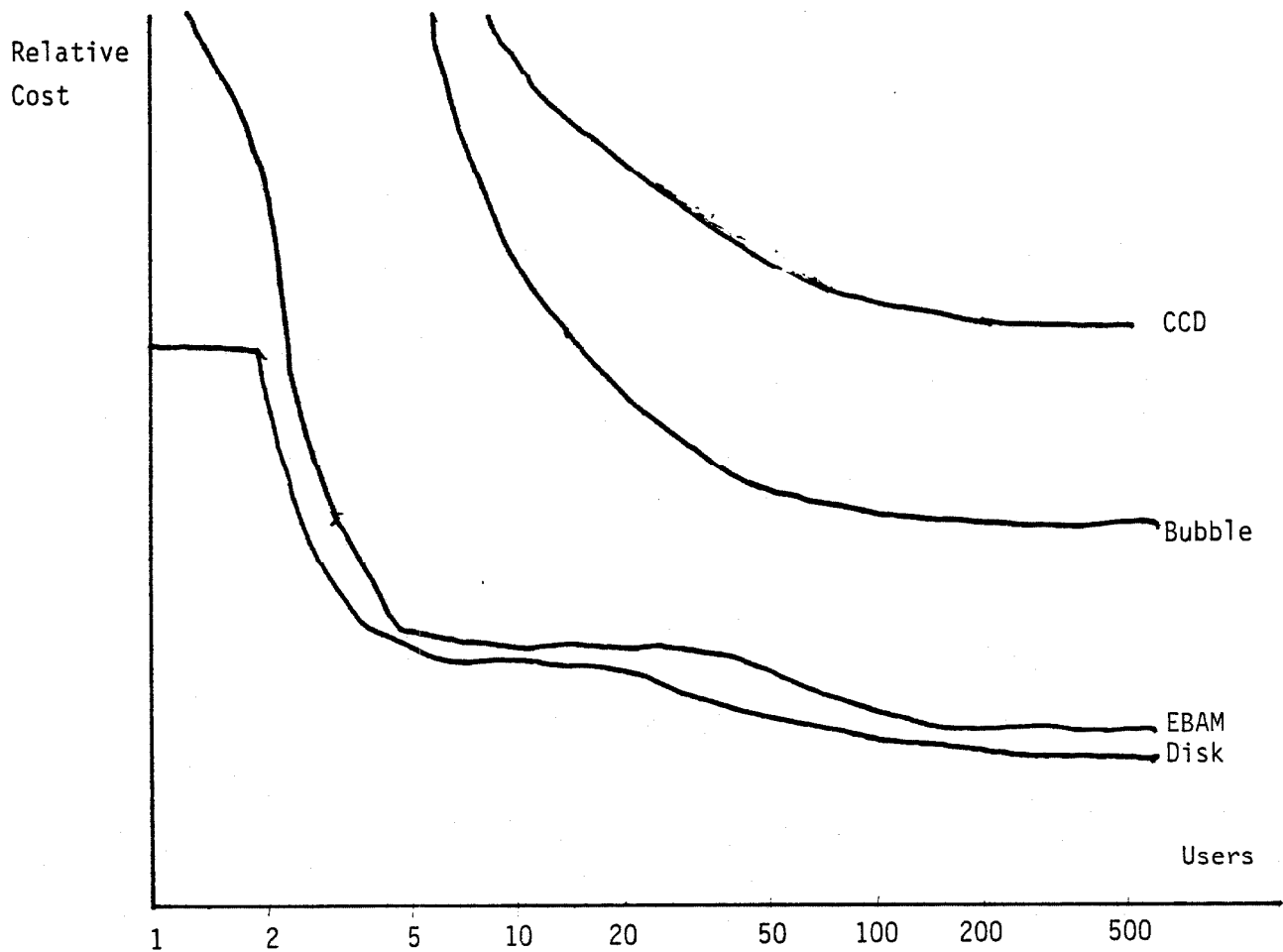


Figure 5-6. Effect of Memory Technology

The cost disadvantage of bubble and CCD technologies increases with larger relation sizes. Indeed, this was the case in all of the architectures with secondary memory which were studied.

Particularly at low page sizes, bubble and CCD offer considerably more accesses per megabyte of store per second. However, it simply appears that the advanced data management system problem better fits the accesses per megabyte per second of disk or EBAM.

Further, it should be noted that this investigation probably has two systematic biases which would make disk look less favorable. First, a page size of 2,000 bytes was used throughout, where examination of Figure 5-5 shows that 4,000 bytes should have been used for disk. Secondly, the capacity projections in Chapter 3 and Figure 5-3 are probably pessimistic.

Will the picture change farther in the future? One can argue not, since disk, bubble, and CCD technologies are driven by the state of the art in photolithographic technique: bubble and CCD for device fabrication, and disk for head fabrication. Thus, with a similar scaling future, it is unlikely that the relative costs of these technologies will change by a factor of two, much less an order of magnitude.

Do three level storage hierarchies make sense for advanced information systems, perhaps using CCD or bubble as the middle level? (*) As the analysis for the smart terminal paging bubble architecture (**) suggests, this case degenerates into paging between the bottom (slowest) levels of the hierarchy. As [Greenfield72] argues, although the pattern of reference to the pages of a given relation can be well optimized, the pattern of reference to relations, and hence to pages on the lowest level of the storage hierarchy, cannot be well predicted.

Finally, since EBAM development has attracted relatively little funding, the conclusion that disk is and will remain the memory technology of choice for advanced data management systems cannot be escaped.

5.5 ARCHIECTURES WITH DISK

Given the conclusions discussed in the prior sections, the analysis can be narrowed from the thirty original candidate architectures to just three: paging disk, logic per head disk, and distributed function disk; all in their centralized variants. A cursory comparison of their cost results shows that it will be a close race.

(*) Since bubble has no performance advantage over disk and a cost disadvantage, the discussion focuses solely on CCD.

(**) See Section 4.2.2.

The paging architecture in fact produces the lowest costs. However, it cannot deliver adequate response time when the relation sizes exceed about 5,000 tuples. The disk itself is the first bottleneck. The CCD and EBAM paging architectures showed that without changing the processor architecture, it is possible to get reasonable response time for up to 100,000 tuples from a paging architecture.

Another way of stating this conclusion is that the effective transfer rate of the disk, accounting for latency and system software, is quite low. The actual rate at which pages can be transferred from a disk is not greater than 50,000 to 100,000 bytes per second, (*) utilizing only a tiny fraction of the bandwidth suggested by the instantaneous transfer rate of the device.

Increasing the instantaneous transfer rate of the disk, either by increasing the bit packing density or by transferring in parallel from several heads, will not significantly improve the matter. To see why, refer to Figure 5-4: even if the slope were zero (infinite transfer rate), latency effects dominate.

(*) Refer to Figure 5-5.

Both the logic per head and distributed function architectures increase the rate at which information can be transferred from a disk storage facility to a processing facility. Thus they can be thought of as ways to implement effective interleaving schemes for disk.

The logic per head architecture does this by moving some of the processing logic into the drive electronics and thus achieves an intimate coupling of processing and storage. The effective transfer rate between the processing elements and the storage elements can be extremely high since each head is transferring at full instantaneous rate.

By contrast, the distributed function architecture maintains the same relationship between the processing element and the storage element as in the paging architecture. It thus also maintains the relatively low effective transfer rate. However, it obtains parallelism in a very simple way.

Both architectures can be readily expanded, which accounts for their success in providing good response times to even very severe demands. However, they differ fundamentally in their costs for very large problems.

The logic per head architecture is $o(n^2/m)$ in space-time product. To see why, consider how it processes the projection of one very large relation, R1, against an even larger relation, R2. If the cardinality (*) of R1 exceeds the size of the CAM, then only the first m tuples of R1 can be loaded into the CAMs, and the entire R2 passed under the heads. This process must be iterated $\text{ceil}(c(R1) / m)$ times.

By contrast, the distributed function architecture operates best when projections are accomplished by merge techniques. As with the paging architectures, if the relations are not already sorted, it is best to presort them. (**) The distributed function architecture thus is fundamentally $(n/m) \cdot \log(n/m)$ in space-time product, where m is the number of processors.

(*) number of tuples. The cardinality of a relation is denoted as $c(R)$.

(**) See [Greenfield72] or [Blasgen76].

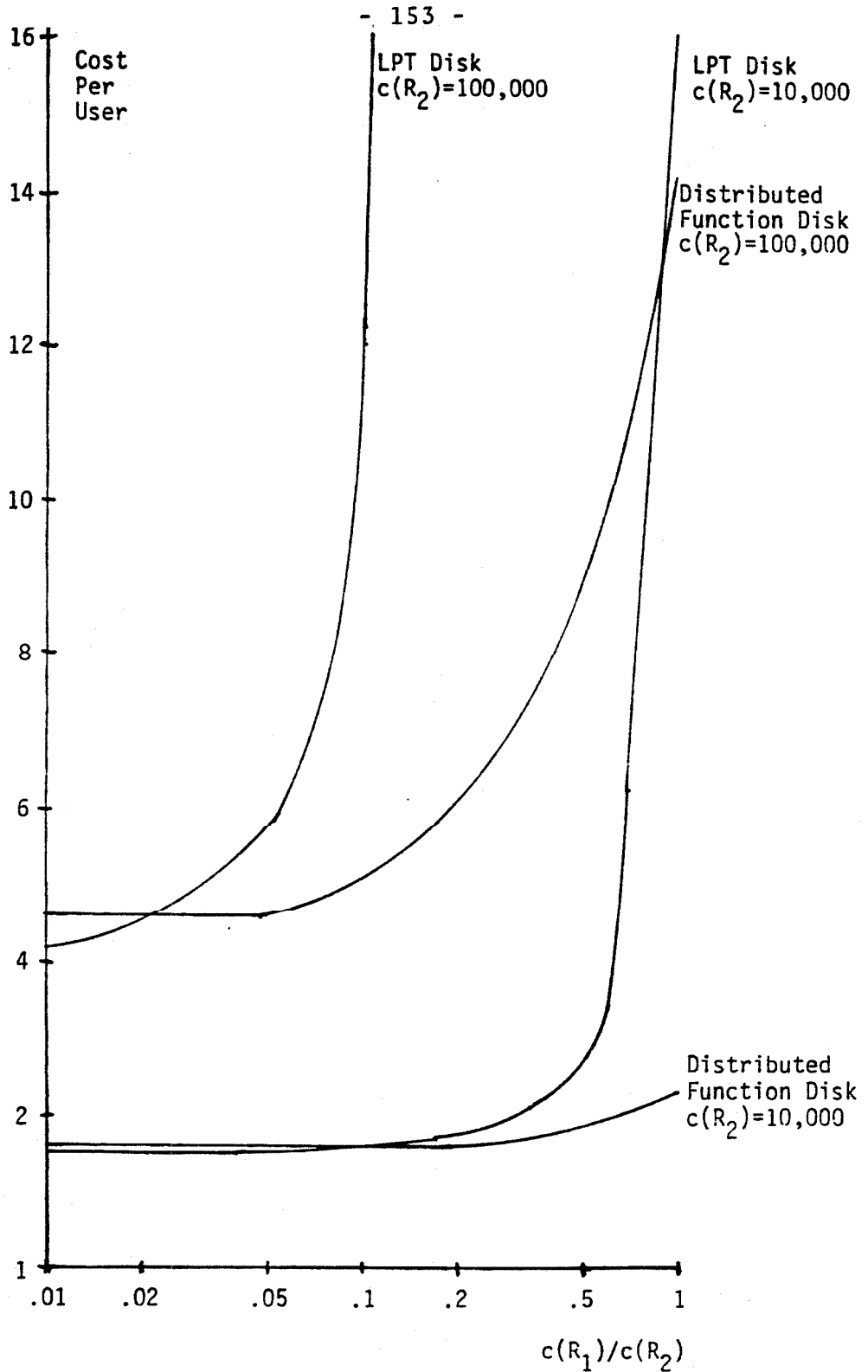


Figure 5-7. Logic Per Head vs. Distributed Function

To explore these phenomena, an additional set of cases was run. With the number of users held fixed at 100, the logic per head and distributed function architectures were subjected to loads in which the cardinality of the larger relation was varied from 500 tuples to 500,000 tuples, while the ratio $c(R1)/c(R2)$ was varied from 1% to 100%. These results are presented on page C-32 for the logic per head architecture and C-33 for the distributed function architecture. Figure 5-7 also presents a summary of the results.

The Logic Per Head architecture does best when the cardinality of the smaller relation is very small compared to the cardinality of the larger relation. Such architectures are effective at matching a small number of patterns against a large number of candidates. (*) However, when the number of patterns to be matched is large, the distributed function architecture produces lower cost results.

Figure 5-8 shows the problem domains in which the distributed function, logic per head, and paging architectures produce lowest cost results. It is worth noting that the cost advantages of logic per track or paging architectures over distributed function

(*) Producing a concordance on selected keywords ("Find all occurrences of the words 'begat', 'beget' or 'begotten' in the Bible") is an example of this problem.

are not great. Thus, by the definition set forth in Section 5.1, the distributed function architecture appears to be the "best."

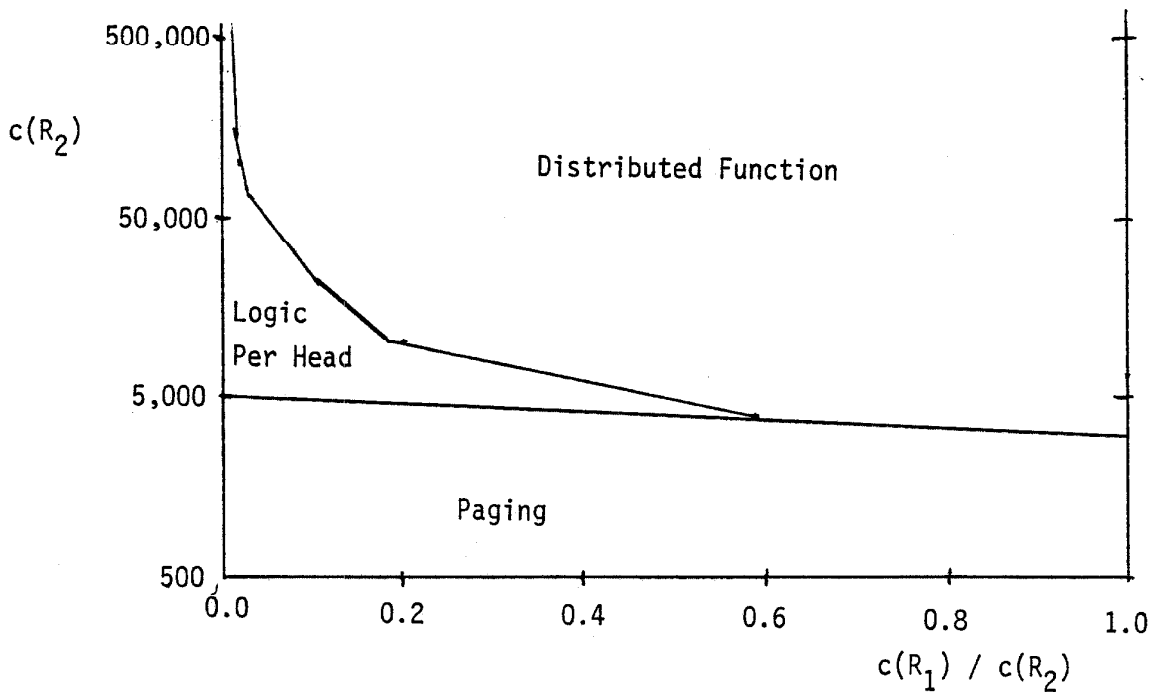


Figure 5-8. Optimal Disk Architectures

5.6 IMPLICATIONS OF DISTRIBUTED FUNCTION

The interconnection bus structure is crucial to the realization of the distributed function architecture, because it is the ultimate bottleneck. The manner in which the distributed function architecture was modeled suggests several requirements for the interconnect structure.

The interconnect needs a broadcast mode of operation. The model assumes that a message requesting that some semantic operator be invoked reaches all processors essentially simultaneously. While an ohmic wire system, such as Ethernet, can do this, it cannot handle the problem of acknowledgement: that is, knowing that the message has been correctly received by all processors.

The interconnect will need to physically connect a large number of processors. This implies that the distance covered by the interconnect will be large: certainly larger than several cabinets.

Besides filling a large volume of space, the interconnect structure must be fast, as it is the bottleneck. A related problem is that of distributing the clock throughout the interconnection network if it is synchronous, or self-timing it.

The merging of results, particularly sorted results, from the processors presents an interesting problem. A simple strategy would be to have each of the m processors sort its subset of tuples, and then transmit blocks of k tuples to one of the processors, which would then do an ordinary merge sort. This processor would need main storage for $k*m$ tuples, and this could be problematical if m were large.

This last requirement suggests that the interconnect structure could be a tournament sort binary tree, with sorting elements in the leaf-to-apex direction, and broadcast elements (*) in the apex-to-leaf direction. This structure could accomplish the merge and broadcast operations. With the 150 nanosecond cycle period assumed in the model, nodes of the tournament sort tree could readily be separated by up to ten meters of wire. Thus it would be possible to configure a physically large system, albeit within one large room.

The implementation of such an interconnect represents an interesting challenge. Would such a structure be interesting to other computing problems besides data management?

The distributed function architecture assumes that the data management problem can be divided into subproblems with fairly high locality of reference, but with little or no locality of reference between subproblems. (This is the case when a projection or join requires that tuples be sorted and redistributed: it is equally likely that any given tuple, when redistributed, will go to any processor.)

(*) i.e., ones which copy the input from the apex direction to both ports in the leaf direction.

Simulation of physical systems with n bodies by integrating the equations of state also have the characteristic of no locality of reference between subproblems. Although the forces found in nature often obey inverse square laws, so that they weaken with distance, the simulation must still consider the interaction of every body with every other body. Even if the numerical value of the force between two bodies is very small, the computational and communications effort is the same as if it were a large value.

Such problems could be treated by assigning bodies or grid points to processors, and having each processor compute an aggregate effective state for the n/m bodies. (This is the subproblem with high locality of reference). Then, each processor would broadcast its aggregate state to the other $m-1$ processors, each of which would compute the forces between its n/m bodies and the input aggregate state to determine the next state of each body. This operation has no locality of reference: like the data management problem, it is characterized by every-to-every communication.

Matrix operations similarly have an every-to-every nature. In the course of matrix inversion, every element will interact with every other element. The distributed function architecture could handle matrix problems by assigning one or more block

submatrices to each processor. The submatrix problem then has very high locality of reference. However, the submatrix operations to yield the full matrix result will require every-to-every communication.

Complex pattern recognition problems could be solved on the distributed function architecture by assigning each processor one or more candidate patterns to match for. The input pattern to be classified is broadcast to all m processors, each of which computes a degree of fit to the pattern(s) assigned to it. The degree of fit values could then be merged to ascertain not only the most probable match, but the next k most likely matches. These could be put through a context-recognition algorithm.

Thus, the distributed function architecture is likely to be interesting to problem domains besides advanced data management.

5.7 DIRECTIONS FOR FUTURE RESEARCH

The selection of trial architectures and the technology forecast used to develop cost functions represents nothing more than attempt to do a set of self-consistent comparisons as far out in time as possible without becoming completely speculative. It would thus be of considerable interest to repeat the same

basic threat of investigation at some time in the future, taking into account actual and anticipated developments in technology and in architecture. It might be quite timely to instigate such an investigation in 1986, as the technology forecast discussed herein expires, and to use 1992 as a target year. Even if the results turned out to be identical, such an investigation would not be without merit.

As is inevitably the case, there was not time in this study to fairly investigate all of the possible parameters. For example, all of the clustered cases used a cluster size of 5: it would be interesting to see how sensitive some of the conclusions are to varying the cluster size.

The study could also be extended to take more account of the man-machine interface. All of the cases presented in this work ignored the issue by assuming an essentially zero cost, zero time interface between the user's terminal and the computing system. While this is neither reasonable nor realistic, it did reduce the scope of the problem, since a data communications environment is at least as complex a problem as a data base environment. However, as discussed in [Neches78] and numerous other references, the modeling techniques which formed the basis for the present study can be readily applied to a broader range of systems.

A final direction of investigation lies in applying the modeling techniques described to the problem of the best hardware architecture for implementation of other kinds of systems, particularly those whose functionality is fairly newly conceived, so that an intuitive approach to understanding their performance may not even be possible. A problem of this sort which promises to have considerable importance in the application of computer technology to the problems of everyday human experience lies in the so-called "expert" systems: systems which incorporate rules of inference and interaction based on the experience of human experts, but in limited domains of discourse. Expert system technology, coupled with vastly improved language processing and data base accessibility, could become an important factor in the computer systems which support all manner of private and public enterprise.

CHAPTER 6 REFERENCES AND BIBLIOGRAPHY

[Astrahan76]

Astrahan, M. M., et al., "System R: A Relational Approach to Database Management" in ACM Transactions on Database Systems, 1, 2, pp. 97 - 137, 1976.

[Bell71]

Bell, C. Gordon and Alan Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971.

[Bhandenkar78]

Bhandenkar, D. P., "Dynamic MOS Memories: Serial or Random Access," Comcon78 Digest of Papers, IEEE, New York, 1978, pp. 162 - 164

[Bigelow73]

Bigelow, R. H., et al., "Specialized Languages: An Application Methodology," Proceedings of the 1973 National Computer Conference, IFIPS Press, New York, 1973.

[Blasgen76]

Blasgen, M. W. and K. P. Eswaran, On the Evaluation of Queries in a Relational Data Base System, IBM Research Report RJ 1745, 1976.

[Chen76]

Chen, W. F., A Performance Study of the CASSM System, Master's Thesis, Department of Electrical Engineering, University of Florida, 1976.

[Codd70]

Codd, E. F., "A Relational Model for Large Shared Data Banks," ACM, 13, 6, pp. 377 - 387, June 1970.

[Copeland74a]

Copeland, G. P., A Cellular System for Non-Numeric Processing, Technical Report No. 1, CASSM Project, University of Florida, Gainesville, 1974.

[Date79]

Date, C. J., Introduction to Data Base Management Systems, Second Edition, Addison-Wesley, Reading, 1975; and Third Edition, Addison-Wesley, Reading, 1982; and Volume II, Addison-Wesley, Reading, 1983.

[DeWitt83]

Dewitt, D. J., "Database Machines" in Proceedings of the Tenth International Symposium on Computer Architecture, IEEE, Stockholm, June 1983 (in press).

[Durniak79]

Durniak, A., "8-inch Hard Disks Set to Go," Electronics, 52, 13, (June 21, 1979), pp. 82 - 84.

[Early78]

Early, J. M., Limitations and Alternatives in Future Silicon Technology, IEEE, New York, 1978.

[Elec78]

Electronics, July 6, 1978, p. 90.

[Gomberg73]

Gomberg, S., The REL Command Language, REL Project Report #8, California Institute of Technology, Pasadena, 1973.

[Gordon67]

Gordon, W. J. and G. F. Newell, "Closed Queueing Systems with Exponential Servers," Operations Research, 15, pp. 254 - 265 (1967).

[Greenfeld72]

Greenfeld, N., Computer Systems Support for Data Analysis, PhD Thesis, California Institute of Technology, Pasadena, 1972.

[Guidry78]

Guidry, M. "CCD" in Early, J. M. (ed.), Limitations and Alternatives in Future Silicon LSI Technology, IEEE, New York, 1978.

[Healy72]

Healy, L. D., K. L. Doty, and G. J. Lipovski, "The Architecture of a Context-Addressed Segment Sequential Storage," Proceedings of the 1972 Fall Joint Computer Conference, 41, Vol. II, AFIPS Press, Montvale NJ, pp. 691 - 701.

[Hess80]

Hess, G. D., A Software Development System, PhD Thesis, California Institute of Technology, Pasadena, 1980.

[Hoeneisen72]

Hoeneisen, B. and C. A. Mead, "Fundamental Limitations in Microelectronics, Part I - MOS Technology," Solid State

Electronics, 15, pp. 819 - 829; and "Fundamental Limitations in Microelectronics, Part II - Bipolar Technology," Solid State Electronics, 15, pp. 891 - 897 (1972).

[Hu78]

Hu, H. L., "Bubbles - A New Magnetic Solid State Technology," Comcon78 Digest of Papers IEEE, New York, 1978, pp. 165 - 166.

[IBM72]

IBM Corporation, OS PL/I Checkout and Optimizing Compilers: Language Reference Manual, SC33-0009-2, Third Edition, Hursley, UK, 1972.

[IBM78]

IBM Corporation, IBM System/38 Technical Developments, IBM Form No. G580-0237, 1978.

[Jackson57]

Jackson, J. R., "Networks of Waiting Lines," Operations Research, 5, pp. 518 - 521 (1957).

[Keyes75]

Keyes, R. W., "Physical Limitations in Digital Electronics," Proceedings of the IEEE, 63, pp. 740 - 767 (May 1975).

[Keyes78]

Keyes, R. W., "Physical Limitations on Computer Devices," Comcon78 Digest of Papers, IEEE, New York, 1978, pp. 294 - 296.

[Kleinrock76]

Kleinrock, L. R., Queueing Systems Volume II: Computer Applications, J. Wiley, New York, 1976.

[Lin76]

Lin, C. S., D. C. P. Smith and J. M. Smith, "The Design of a Rotating Associative Memory for Relational Database Applications," ACM Transactions on Database Systems, 1, 1, March 1976, pp. 53 - 65.

[Lipovski78]

Lipovski, G. J., "Architectural Features of CASSM: A Context Addressed Segment Sequential Memory," Proceedings of the Fifth Annual Symposium on Computer Architecture, Palo Alto, April 1978, pp. 31 - 38.

[Locanthi77]

Locanthi, B. N., An Associative Main Memory, Display File No. 592, Department of Computer Science, California Institute of Technology, Pasadena, March, 1977; and An Associative Memory Chip, Display File No. 1066, Department of Computer Science, California Institute of Technology, Pasadena, October, 1977; and Associative Memories: An Appraisal, Display File No. 1863, Department of Computer Science, California Institute of Technology, Pasadena, July, 1978.

[Locanthi78]

Locanthi, B. N., LAP: A SIMULA-Based Graphics Package for VLSI Design, SSP Report #1862, Department of Computer Science, California Institute of Technology, Pasadena, 1978.

[Martin76]

Martin, J. M., Principles of Data Base Management, Prentice-Hall, New York, 1976.

[Mead79]

Mead, C. A. and L. A. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Massachusetts, 1980.

[Metcalf75]

Metcalf, R. M. and D. Boggs, Ethernet, Xerox Palo Alto Research Center, 1975.

[Mohsen79]

Mohsen, A., "Devices and Circuits for VLSI," Proceedings of the Industrial Associates Conference on Very Large Scale Integration, California Institute of Technology, Pasadena, 22 - 24 January 1979.

[Moore71]

Moore, C. G., Network Models for Large-Scale Timesharing Systems, Technical Report No. 71-1, Department of Industrial Engineering, University of Michigan, Ann Arbor, April 1971.

[Muntz72a]

Muntz, R. R. and F. Basket, Open, Closed, and Mixed Networks of Queues with Different Classes of Customers, Technical Report No 33, Stanford Electronics Laboratories, Stanford, August 1972.

[Muntz72b]

Muntz, R. R., Poisson Departure Processes and Queueing Networks, IBM Research Report RC4145, December 1972.

[Muntz74]

Muntz, R. R. and J. Wang, "Efficient Computational Procedures for Closed Queueing Network Models," Proceedings of the Seventh

Hawaii International Conference on System Science, Honolulu, pp. 33 - 36, January 1974.

[Neches76]

Neches, P. M., Processor Load Projection, TTI-76-0061-00, Transaction Technology Inc., Los Angeles, May 1976.

[Neches78]

Neches, P. M., Architectures for the Requirements and Technologies of the 1980's, (Thesis proposal), Display File #1646, Department of Computer Science, California Institute of Technology, Pasadena, May 1978.

[Noyce77]

Noyce, R. N., "Microelectronics," Scientific American, September 1977.

[Ozkarahan75]

Ozkarahan, E. A., S. A. Schuster, and K. C. Smith, "RAP - An Associative Processor for Data Base Management," Proceedings of the 1975 National Computer Conference, 45 AFIPS Press, Montvale NJ, pp. 379 - 387.

[Ozkarahan76]

Ozkarahan, E. A., An Associative Processor for Relational Data Bases - RAP, PhD Thesis, University of Toronto, 1976.

[Ozkarahan77]

Ozkarahan, E. A. and K. C. Sevick, "Analysis of Architectural Features for Enhancing the Performance of a Database Machine," ACM Transactions on Database Systems, 2, 4, December 1977, pp. 297 - 316.

[Pashley78]

Pashley, R., "N-Channel" in Early, J. M. (ed.), Limitations and Alternatives in Future Silicon LSI Technology, IEEE, New York, 1978.

[Rem78]

Rem, M. and C. A. Mead, The Cost and Performance of VLSI Computing Structures, Display File No. 1584, Department of Computer Science, California Institute of Technology, Pasadena, 1978.

[Schuster76]

Schuster, S. A., E. A. Ozkarahan, and K. C. Smith, "A Virtual Memory System for a Relational Associative Processor," Proceedings of the 1976 National Computer Conference, 45, AFIPS Press, Montvale NJ, pp. 855 - 862.

[Shaw79]

Shaw, D. E., A Hierarchical Architecture for the Parallel Evaluation of Relational Algebraic Database Primitives, Technical Report STAN-CS-79-778, PhD Thesis, Stanford University, 1979.

[Share76]

Share Incorporated, Data Processing in 1980-1985: A Study of the Potential Limitations to Progress, Wiley-Interscience, New York, 1976.

[Slotnick70]

Slotnick, D. L., "Logic Per Track Devices," Advances in Computers, 10, Academic Press, New York, 1970, pp. 291 - 296.

[Smith78]

Smith, D. O., "Electron Beam Accessed Memory," Comcon78 Digest of Papers, IEEE, New York, pp. 167 - 169, 1978.

[Smith79]

Smith, D. C. P. and J. M. Smith, "Relational Data Base Machines," Computer, 12, 3, pp. 28 - 39, March 1979.

[Solovits73]

Solovits, P., A Specialized Language Implementation Facility, PhD Thesis, California Institute of Technology, Pasadena, 1973.

[Su75]

Su, S. Y. W. and G. J. Lipovski, "CASSM: A Cellular System for Very Large Data Bases," Proceedings of the First International Conference on Very Large Data Bases, September 1975, pp. 456 - 472.

[Su77]

Su, S. Y. W., "Associative Programming in CASSM and Its Applications," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan, 6-8 October 1977, pp. 213 - 228.

[Su78]

Su, S. Y. W. and A. Emam, "CASDAL: CASSM's Data Language," ACM Transactions on Data Base Systems, 3, 1, March 1978, pp. 57 - 91.

[Sutherland78]

Sutherland, I. E. and C. A. Mead, "Microelectronics and Computer Science," Scientific American, September 1977.

[Tandem76]

Tandem Computers Incorporated, Tandem 16 Programming Manual, Product No. T16/6001, Cupertino, May 1976.

[Thompson68]

Thompson, F. B., "The Organization Is the Information," Journal of American Documentation, 19, 3, July 1968, pp. 305 - 308.

[Thompson74a]

Thompson, F. B., The REL Paging Services, REL Project Report No. 18, California Institute of Technology, Pasadena, 1974.

[Thompson74b]

Thompson, F. B., The REL I/O Services REL Project Report No. 19, California Institute of Technology, Pasadena, 1974.

[Thompson74c]

Thompson, F. B., et al., The REL Antimated Film Language, REL Project Report No. 12, California Institute of Technology, Pasadena, 1974.

[Thompson75]

Thompson, B. H. and F. B. Thompson, "Practical Natural Language Processing," Advances in Computers, 13, pp. 109 - 168, 1975.

[Turn74]

Turn, R., Computers in the 1980's, Columbia University Press, New York, 1974.

[Ullman80]

Ullman, Jeffery D., Principles of Database Systems, Computer Science Press, Potomoc, Maryland, 1980.

[Wu78]

Wu, I. C., "VLSI and Mainframe Computers," Comcon78 Digest of Papers, IEEE, New York, 1978, pp. 25 - 30.

[Yu80]

Yu, K. I., Communicative Databases, PhD Thesis, California Institute of Technology, 1980.

APPENDIX A EXAMPLE OF THE MODEL

<u>Page</u>	<u>Routine</u>
A- 2	Declarations: Queueing Model Data Structures
A- 4	Declarations: Global Variables
A- 5	Driver Routine
A- 7	Default Procedure
A- 8	Heuristic Node Grow Routine
A- 9	Cost Function
A-10	Expand Utility
A-11	Config Procedure: Node Definitions
A-12	Config Procedure: Common Flow Subroutines
A-13	Config Procedure: Flow Definitions
A-15	Config Procedure: Transaction Definitions
A-16	Queueing Model: Add a new Node
A-17	Queueing Model: Add a new Flow
A-18	Queueing Model: Add a Step to a Flow
A-19	Queueing Model: Copy a Flow
A-20	Queueing Model: Add a new Transaction
A-21	Queueing Model: Add a Flow to a Transaction
A-22	Queueing Model: Driver
A-23	Queueing Model: Calculate the Load per Node for each step
A-24	Queueing Model: Aggregate Node Load by Transaction Type
A-25	Queueing Model: Calculate Node Utilization and Waiting
A-27	Queueing Model: Calculate Time per Step
A-28	Queueing Model: Aggregate Time per Flow
A-29	Queueing Model: Aggregate Time per Transaction
A-31	Queueing Model: Aggregate Time per Node
A-32	Report Writer: Driver
A-33	Report Writer: Node Reports
A-34	Report Writer: Flow Detail Report
A-35	Report Writer: Transaction Detail Report

/* REL CCNFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 */ 000100C0

```

STMT LEVEL NEST          /* REL CCNFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 */
                          000100000
1      FREQ: PROCEDURE OPTIONS (MAIN);          00010010
2      /* DATA STRUCTURE FOR TRANSACTION */    00010020
        DECLARE
        1 TRANSACTION (50);
          2 NAME CHARACTER (32), /* INPUT */
          2 WEIGHT BINARY FLOAT, /* INFLT */
          2 FIRSTFLOW BINARY FIXEC (31),
          2 LASTFLOW BINARY FIXEC (31),
          2 TIME BINARY FLOAT,
          2 SIGMA BINARY FLOAT,
        1 THISTXN LIKE TRANSACTION BASED (FIXN),
        /* DATA STRUCTURE FOR TRANSACTION FLOWS */
        1 TXNFLOW (250),
          2 FLCW POINTER,
          2 REPTITIONS BINARY FLOAT,
          2 TIME BINARY FLOAT,
          2 SIGMA BINARY FLOAT,
        1 THISTXNFLOW LIKE TXNFLOW BASED (FIXNFLOW),
                          00010110
                          00010120
                          00010130
                          00010140
                          00010150
                          00010160
                          00010170
                          00010180
    
```

STMT LEVEL NEST

```

/* DATA STRUCTURE FOR FLOWS */
00010190
00010200
00010210
00010220
00010230
00010240
00010250

1 FLOW (100),
  2 NAME CHARACTER(32) /*INPLT*/ ,
  2 FIRSTSTEP BINARY FIXED (31),
  2 LASTSTEP BINARY FIXED (31),
  2 TIME BINARY FLOAT,
  2 SIGMA BINARY FLOAT,

00010260

1 THISFLOW LIKE FLOW BASED (PFLCM),

/* DATA STRUCTURE FOR STEPS */
00010270
00010280
00010290
00010300
00010310
00010320
00010330
00010340
00010350

1 STEP (400),
  2 NODE POINTER,
  2 DATASIZE BINARY FLOAT, /* INPUT */
  2 MULTIPLIER BINARY FLOAT, /* INPUT */
  2 PAGES BINARY FLOAT,
  2 TIME BINARY FLOAT,
  2 SIGMA BINARY FLOAT,
  2 LOAD BINARY FLOAT,

00010360

1 THISSTEP LIKE STEP BASED (FSTEP),

/* DATA STRUCTURE FOR NODES */
00010370
00010380
00010390
00010400
00010410
00010420
00010430
00010440
00010450
00010460
00010470
00010480
00010490
00010500

1 NODE (100),
  2 NAME CHARACTER (32), /* INPLT */
  2 PAGESIZE BINARY FLOAT, /* INPUT */
  2 PAGERATE BINARY FLOAT, /* INPUT */
  2 USERS BINARY FLOAT, /* INPUT */
  2 SERVERS BINARY FLOAT, /* INPUT */
  2 QUELETYPE BINARY FLOAT, /* INPUT */
  2 LOAD (50) BINARY FLOAT,
  2 CAPACITY BINARY FLOAT,
  2 UTILIZATION BINARY FLOAT,
  2 TIME BINARY FLOAT,
  2 MEANFACTOR BINARY FLOAT,
  2 SIGMAFACTOR BINARY FLOAT,

00010510
  
```

STMT LEVEL NEST

```

/* GLOBAL VARIABLES */
(MAX TIME, MIN TIME, WEIGHTED MEAN TIME) REAL,
(WNODES, #FLOWS, #IXNS, #PAGE, #STEP, #TXNFLOW)
BINARY FIXED (31) INITIAL (0),
( REREAD, EOF) BIT(1) INITIAL ('0'B),
ARRIVAL_RATE BINARY FLCAT INITIAL (C-01),
DATE BUILT IN,
DATE6 CHARACTER (6),
YEAR CHARACTER (2) DEFINED CATE6 POSITION(1),
MONTH PICTURE '99' DEFINED CATE6 POSITION(3),
DAY CHARACTER (2) DEFINED CATE6 POSITION(5),
MONTHTABLE (12) CHARACTER (9) VARYING INITIAL
('JANUARY', 'FEBRUARY', 'MARCH', 'APRIL', 'MAY', 'JUNE',
'JULY', 'AUGUST', 'SEPTEMBER', 'OCTOBER', 'NOVEMBER', 'DECEMBER'),
DATESTRING CHARACTER (20),
BUFFER CHARACTER (80),
TRUE BIT (1) INITIAL ('1'B),
FALSE BIT(1) INITIAL ('0'B);

```

```

/* END OF DECLARATIONS */

```

```

/* INITIALIZATION CODE FOR PAGE HEADINGS */

```

```

ON ENDPAGE (SYSPRINT) BEGIN;
  #PAGE = #PAGE + 1;
  IF #PAGE > 1 THEN PUT PAGE;
  PUT EDIT ('REL CONFIGURATION MODEL - PL/I IMPLEMENTATION ',
           'VERSION 4.0 OF 22-AUG-1975 - RUN DATE: ', DATESTRING,
           ' - PAGE ', #PAGE) (A,A,A,A,F(4));
  PUT SKIP (3);
END;

```

```

/* INITIALIZE DATESTRING */
DATE6 = /* SYSTEM */ DATE;
DATESTRING = DAY || ' ' || MNCHTTABLE(MNTH) || ' ' || YEAR;
OPEN FILE(SYSPRINT) LINESIZE(132);

```



```

STMT LEVEL NEST
14 1 /* DRIVER: MAIN PROGRAM WHICH DRIVES OUTPUT CASES */
    DECLARE
    ICLASS_SIZE, RELATION_SIZE, #CLASSES, #RELATICS, #SEATTLEMS,
    PAGE_SIZE, RULE_SIZE, RULES, DEFINITICS, PRORDEF, SEATTLEM,
    #TRACKS, TRACK_SIZE, HOST_MIPS, UP_MIPS, RPM, SSKRATE, SCRATE, MISRATE,
    BUS_WIDTH, BUS_CYCLE, GROUP_SIZE, #GROUPS, PACKET_SIZE, BASED,
    LATERAL, LGRRATE, CVRRATE, EANDRDT, PROP_DELAY,
    SYSUSERS, #HOSTS, #DRIVES, #CHANNELS, PAGEFRAMES, BIN FLOAT:
    DCL (I, J, ITRIES, IN, INN, MAXTRIES) BIN FIXED (31),
    (C, X, Y, WORST) REAL, GRCT_FLAG BIT(1),
    PICC_PICTURE -----S*,
    CUTMATRIX (0:10, 0:10) CHAR (12) INIT (('121')) *;;

15 1 MAX_TIME = 15; MIN_TIME = 1; MAXTRIES = 50;

16 1 DO Y = 0.0 TC 0.25 BY 0.05;

19 1 I = 0; SIGNAL ENDPAGE (SYSFRINT);
    PUT EDIT (I, 'USERS REL SIZE XBASED %LATERAL',
    , #HOSTS #DRIVES ITERATIONS') (A,A);
    PUT SKIP (2);
    DO X = 500, 1E3, 2E3, 5E3, 1E4, 2E4, 5E4, 1E5, 2E5, 5E5;
    CALL DEFAULT;
    BASED = Y; LATERAL = Y/2;
    GROW_FLAG = TRUE;
    RELATION_SIZE = X;
    I = I + 1; J = 0; PICC = X; CUTMATRIX (I,J) = PICC;
    DO SYSUSERS = 1, 2, 5, 10, 20, 50, 100, 200, 500;
    J = J + 1; PICC = SYSUSERS; CUTMATRIX (0,J) = PICC;
    DO ITRIES = 1 TO MAXTRIES;
    CALL CONFIG;
    IF WEIGHTED_MEAN_TIME < MIN_TIME ( GROW_FLAG THEN DO;
    CUTMATRIX (I,J) = '100 BIG';
    GO TO NEXT_CASE;
    END;
    ELSE IF WEIGHTED_MEAN_TIME <= MAX_TIME THEN DO;
    C = COST;
    PICC = C; CUTMATRIX (I,J) = PICC;
    IF SYSUSERS = 100 & RELATION_SIZE = 1E3
    THEN CALL REPORT /* FOR THE GORY DETAILS */;
    ELSE ;
    GO TO NEXT_CASE;
    END;
    ELSE DO; /* FIND /AC EXPAND THE WORST-CASE NCCLE */
    GROW_FLAG = FALSE;
    IN = 1; WORST = ACCCE(1).TIME;
    DO INN = 2 TC ACCDES;
    IF NODE(INN).TIME > #CRST THEN DO;
    IN = INN;
    #CRST = ACCCE(INN).TIME;
    END;
    END;

20 1 I = 0; SIGNAL ENDPAGE (SYSFRINT);
22 1 PUT EDIT (I, 'USERS REL SIZE XBASED %LATERAL',
    , #HOSTS #DRIVES ITERATIONS') (A,A);
23 1 PUT SKIP (2);
24 1 DO X = 500, 1E3, 2E3, 5E3, 1E4, 2E4, 5E4, 1E5, 2E5, 5E5;
25 1 CALL DEFAULT;
26 1 BASED = Y; LATERAL = Y/2;
27 1 GROW_FLAG = TRUE;
28 1 RELATION_SIZE = X;
29 1 I = I + 1; J = 0; PICC = X; CUTMATRIX (I,J) = PICC;
30 1 DO SYSUSERS = 1, 2, 5, 10, 20, 50, 100, 200, 500;
31 1 J = J + 1; PICC = SYSUSERS; CUTMATRIX (0,J) = PICC;
32 1 DO ITRIES = 1 TO MAXTRIES;
33 1 CALL CONFIG;
34 1 IF WEIGHTED_MEAN_TIME < MIN_TIME ( GROW_FLAG THEN DO;
35 1 CUTMATRIX (I,J) = '100 BIG';
36 1 GO TO NEXT_CASE;
37 1 END;
38 1 ELSE IF WEIGHTED_MEAN_TIME <= MAX_TIME THEN DO;
39 1 C = COST;
40 1 PICC = C; CUTMATRIX (I,J) = PICC;
41 1 IF SYSUSERS = 100 & RELATION_SIZE = 1E3
42 1 THEN CALL REPORT /* FOR THE GORY DETAILS */;
43 1 ELSE ;
44 1 GO TO NEXT_CASE;
45 1 END;
46 1 ELSE DO; /* FIND /AC EXPAND THE WORST-CASE NCCLE */
47 1 GROW_FLAG = FALSE;
48 1 IN = 1; WORST = ACCCE(1).TIME;
49 1 DO INN = 2 TC ACCDES;
50 1 IF NODE(INN).TIME > #CRST THEN DO;
51 1 IN = INN;
52 1 #CRST = ACCCE(INN).TIME;
53 1 END;
54 1 END;
55 1 I = 0; SIGNAL ENDPAGE (SYSFRINT);
56 1 PUT EDIT (I, 'USERS REL SIZE XBASED %LATERAL',
57 1 , #HOSTS #DRIVES ITERATIONS') (A,A);
58 1 PUT SKIP (2);
59 1 DO X = 500, 1E3, 2E3, 5E3, 1E4, 2E4, 5E4, 1E5, 2E5, 5E5;
60 1 CALL DEFAULT;
61 1 BASED = Y; LATERAL = Y/2;
62 1 GROW_FLAG = TRUE;
63 1 RELATION_SIZE = X;
64 1 I = I + 1; J = 0; PICC = X; CUTMATRIX (I,J) = PICC;
65 1 DO SYSUSERS = 1, 2, 5, 10, 20, 50, 100, 200, 500;
    J = J + 1; PICC = SYSUSERS; CUTMATRIX (0,J) = PICC;
    DO ITRIES = 1 TO MAXTRIES;
    CALL CONFIG;
    IF WEIGHTED_MEAN_TIME < MIN_TIME ( GROW_FLAG THEN DO;
    CUTMATRIX (I,J) = '100 BIG';
    GO TO NEXT_CASE;
    END;
    ELSE IF WEIGHTED_MEAN_TIME <= MAX_TIME THEN DO;
    C = COST;
    PICC = C; CUTMATRIX (I,J) = PICC;
    IF SYSUSERS = 100 & RELATION_SIZE = 1E3
    THEN CALL REPORT /* FOR THE GORY DETAILS */;
    ELSE ;
    GO TO NEXT_CASE;
    END;
    ELSE DO; /* FIND /AC EXPAND THE WORST-CASE NCCLE */
    GROW_FLAG = FALSE;
    IN = 1; WORST = ACCCE(1).TIME;
    DO INN = 2 TC ACCDES;
    IF NODE(INN).TIME > #CRST THEN DO;
    IN = INN;
    #CRST = ACCCE(INN).TIME;
    END;
    END;

```

STMT LEVEL NEST

```

66 1 5
68 1 6
69 1 6
70 1 6
71 1 5
72 1 4
    IF GROW(LIN) THEN CC;
    OUTMATRIX(I,J) = 'TCC SMALL';
    GO TO NEXT_FCH;
    END;
    END;
    END;

73 1 3
    NEXT_CASE: PUT SKIP EDIT (SYSUSERS, RELATION_SIZE, BASED, LATERAL,
    HOST_MIPS, #HOSTS, #DRIVES, #TRIES) (I9)(F10.2));
    END;
    END;
    NEXT_ROW:
    END;
    END;

/* OUTPUT THE COST MATRIX */
77 1 1
78 1 1
    SIGNAL ENDPAGE (SYSPRINT);
    PUT EDIT ('CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) *
    'VERSUS RELATION SIZE IN TUPLES (ROW)') (SKIP(2),A,A);
79 1 1
    PUT EDIT ((OUTMATRIX(I,J) DC J = 0 TO 10) DO I = 0 TO 10)
    (SKIP(2),(11)(A(12)));
80 1 1
    END;

```

```

00020430
00020440
00020450
00020460
00020470
00020480

00020490
00020492
00020500
00020510
00020520

00020530
00020540
00020542
00020544
00020550
00020560
00020565

```


STMT LEVEL NEST

```

113 1   DCL GRW ENTRY (BIN FIXED(31)) RETUFAS (BIT(1));          00020810
114 1   GRCK: PROCEDURE (IN) RETURNS (BIT(1));                 00020820
115 2   DCL (IN, NEWMODULES) BIN FIXED (31),
      OK BIT(1) INIT('0'B), CANTLO BIT(1) INIT ('1'B),
      GCGRCNODE (7) LABEL;                                     00020830
      00020840
      00020850
116 2   IF IN < 1 | IN > 7 THEN SIGNAL ERFCR;                  00020860
118 2   GC IO GCGRCNODE (IN);                                   00020870
119 2   GCGRCNODE(1); /* TERMINAL PROCESSOR */
      GCGRCNODE(5); /* REMOTE PROCESSOR */
      #HOSTS = #HOSTS + 1;
120 2   IF #HOSTS > 3 THEN DCI;
122 2   #HOST_MIPS = 4*#HOST_MIPS;
123 2   #HOSTS = 1;
124 2   END;
125 2   IF #HOST_MIPS > 16 THEN RETURN (CANTCC); ELSE RETURN (CK);

A-8
128 2   GCGRCNODE(2); /* LOCAL DISK */
      GCGRCNODE(6); /* REMOTE DISK */
      #DRIVES = #DRIVES + 1;
129 2   IF #DRIVES > 8 THEN RETURN (CANTCC); ELSE RETURN (OK);
132 2   GCGRCNODE(3); /* LOCAL CTLR */
      GCGRCNODE(7); /* REMOTE CTLF */
      RETURN (CANTDD);
133 2   GCGRCNODE(4); /* ETHERNET */
      BANDWIDTH = BANDWIDTH + 1EC;
134 2   PROP_DELAY = 1.25E-6 * BANDWIDTH;
135 2   IF BANDWIDTH > 10E6 THEN RETURN (CANTDD); ELSE RETURN (OK);
138 2   END GRW;
    
```

```

SYMT LEVEL NEST
139 1 CCL CCST ENTRY RETURNS (REAL); 00021090

140 1 COST: PROCEDURE;
141 2 DCL DBSIZE, #CHIPS) REAL;
142 2 DBSIZE = 10 * CLASS_SIZE * #CLASSES
      + 20 * RELATION_SIZE * ARELATIONS
      + RULESIZE * (RULES + DEFINITIONS);
143 2 DBSIZE = DBSIZE * SYSUSERS;
144 2 #CHIPS = 8 * DBSIZE / 256E3;
145 2 #DISKS = CEIL (DBSIZE / 1E5);
146 2 RETURN ((
      3600 + 13500*HOST_MIPS*#HOSTS
      + 1500 * #CHIPS / 32
      + 8E3 * #DISKS)/SYSUSERS + 1E3);
147 2 END COST;

00021100
00021110
00021120
00021130
00021140
00021145
00021150
00021160
00021170
00021180
00021190
00021200
00021210

```

STMT LEVEL NEST

```

148 1 DCL EXPAND ENTRY (BIN FIXED(31)) RETURNS (REAL); 00030000
149 1 EXPAND: PROCEDURE (IN); 00030010
150 2 DCL IN BIN FIXED (31), RHC_TARGET REAL; 00030020
151 2 PNODE = ADDR(NODE(IN)); 00030030
152 2 /* IS UTILIZATION > 100% THE PROBLEM ? */ 00030040
153 2 IF THISNODE.UTILIZATION > 0.95 THEN 00030050
    RETURN (THISNODE.UTILIZATION*1.CS); 00030060
154 2 /* COULD IT BE QUEUEING DELAYS? */ 00030070
155 2 ELSE IF THISNODE.TIME/THISNODE.MEANFACTOR < MAX_TIME THEN DO: 00030080
    RHO_TARGET = 1 - (THISNODE.TIME / THISNODE.MEANFACTOR) 00030090
    / MAX_TIME; 00030100
157 2 IF RHC_TARGET > 0 & RHO_TARGET < THISNODE.UTILIZATION 00030110
158 2 THEN RETURN (THISNODE.UTILIZATION / RHO_TARGET); 00030120
159 2 ELSE RETURN (2 * THISNODE.TIME/MAX_TIME); 00030130
160 2 END; 00030140
161 2 ELSE RETURN (2 * THISNODE.TIME / MAX_TIME); 00030150
162 2 END EXPAND; 00030160

```

STMT LEVEL NEST

163 1 CCNFIG: PROCEDURE: 00030170

164 2 DECLARE
 NEMNCOE ENTRY (CHAR(32), EIN FLCAT, BIN FLOAT,
 BIN FLOAT, BIN FLCAT, BIN FIXED (31)),
 NEMFLCH ENTRY (CHAR(32)),
 NEMSTEP ENTRY (CHAR(32), BIN FLCAT, BIN FLOAT),
 NEMTXN ENTRY (CHAR(32), EIN FLCAT),
 USEFLOW ENTRY (CHAR(32), EIN FLCAT),
 COPYFLOW ENTRY (CHAR(32));

165 2 #NODES, #FLOWS, #TXNS, #STEP, #MAXFLCH = 0; 00030260

166 2 DECLARE (PGS, GRSIZE, PRGJ) EIN FLCAT; 00040000

167 2 GRSIZE = RULESIZE * (RULES + DEFINITIONS); 00040010
 168 2 #GROUPS = CEIL (SYSUSERS / GROUP_SIZE); 00040015

/* ESTABLISH NODES FOR INDIVIDUAL CENTRAL CONFIGURATION */ 00040020

169 2 CALL MEMNODE ('CENTRAL PROCESSOR', 1, IE6*HOST_MIPS,
 SYSUSERS, #HOSTS, 3); 00040030
 170 2 CALL MEMNODE ('LOCAL DISK ARM', 1, 33, 1, #DRIVES, 2); 00040031
 171 2 CALL MEMNODE ('LOCAL DISK CTRL', PAGESIZE,
 1/(10.5 + PAGESIZE/TRACKSIZE) * (1/(RPM/60))), 1, 1, 2); 00040050
 172 2 CALL MEMNODE ('ETHERNET', C.123, BANDWIDTH, SYSUSERS,
 1, 2); 00040060
 173 2 CALL MEMNODE ('REMOTE PROCESSOR', 1, IE6*UP_MIPS, SYSUSERS,
 #GROUPS, 2); 00040080
 174 2 CALL MEMNODE ('REMOTE DISK', 1, 33, GROUP_SIZE * #DRIVES, 00040100
 2); 00040110
 175 2 CALL MEMNODE ('REMOTE CTRL', PAGESIZE,
 1/(10.5 + PAGESIZE/TRACKSIZE) * (1/(RPM/60))),
 GROUP_SIZE, GROUP_SIZE, 2); 00040130
 00040140
 00040150

```

STMT LEVEL NEST
176 2 DCL LOCALBLOCK ENTRY (BIN FICAT);
177 2 LOCALBLOCK: PROCEDURE (BYTES);
178 3 DCL (PGS, BYTES) BIN FICAT;
179 3 PGS = CEIL (BYTES/PAGESIZE);
180 3 CALL NEWSTEP ('CENTRAL_PROCESSOR', 2000, 1);
181 3 END LOCALBLOCK;
182 2 DCL REMOTEBLOCK ENTRY (BIN FICAT);
183 2 REMOTEBLOCK: PROCEDURE (BYTES);
184 3 DCL (BYTES, PGS) BIN FICAT;
185 3 PGS = CEIL (BYTES / PAGESIZE);
186 3 CALL NEWSTEP ('REMOTE_PROCESSOR', 2000, 1);
187 3 END REMOTEBLOCK;
188 2 DCL INTERPROCESSOR ENTRY (BIN FICAT);
189 2 INTERPROCESSOR: PROCEDURE (BYTES);
190 3 DCL (BYTES, PACKETS, FULLPACKETS, CDDPACKET) BIN FICAT;
191 3 PACKETS = CEIL (BYTES / PACKET_SIZE);
192 3 FULLPACKETS = FLOOR (BYTES / PACKET_SIZE);
193 3 CDDPACKET = BYTES - FULLPACKETS*PACKET_SIZE;
194 3 CALL NEWSTEP ('CENTRAL_PROCESSOR', 2000 * (PACKETS+1), 1);
195 3 IF FULLPACKETS > 0 THEN
196 3 CALL NEWSTEP ('ETHERNET', PACKET_SIZE * 20 + PROP_DELAY,
FULLPACKETS);
197 3 IF PACKETS > FULLPACKETS THEN CALL NEWSTEP ('ETHERNET',
CDDPACKET + 20 + PROP_DELAY, 1);
199 3 CALL NEWSTEP ('REMOTE_PROCESSOR', 2000*(PACKETS+1), 1);
200 3 END INTERPROCESSOR;
00040160
00040170
00040180
00040190
00040200
00040230
00040240
00040250
00040260
00040270
00040290
00040310
00040320
00040330
00040340
00040350
00040360
00040365
00040370
00040375
00040380
00040390
00040400
00040410
00040420
00040430

```



```

STMT LEVEL NEST
/* THE FLOWS */
00040440
201 2 CALL NEWFLOW ('LOAD LOCAL PAGE');
202 2 CALL LOCALBLOCK (PAGESIZE);
00040450
00040460
203 2 CALL NEWFLCH ('LOAD REMOTE PAGE');
204 2 CALL REMOTEBLOCK (PAGESIZE);
205 2 CALL INTERPROCESSOR (PAGESIZE);
00040470
00040480
00040490
206 2 CALL NEWFLCH ('PARSE LOCAL');
207 2 CALL LOCALBLOCK (GRSIZE);
208 2 CALL NEWSTEP ('CENTRAL PROCESSOR', 20000 +
(100*LOG(GRSIZE) + 100) * SENTLEN/11-PROBDEF), 1);
00040500
00040510
00040520
00040530
209 2 CALL NEWFLOW ('PARSE REMOTE');
210 2 CALL INTERPROCESSOR (100);
211 2 CALL REMOTEBLOCK (GRSIZE);
212 2 CALL NEWSTEP ('REMOTE PROCESSOR', 20000 +
(100*LOG(GRSIZE) + 100) * SENTLEN/11-PROBDEF), 1);
00040540
00040550
00040560
00040570
00040580
213 2 CALL NEWFLOW ('LOCAL RECGRC');
214 2 CALL LOCALBLOCK (100);
00040590
00040600
215 2 CALL NEWFLCH ('REMOTE RECFD');
216 2 CALL INTERPROCESSOR (100);
217 2 CALL REMOTEBLOCK (100);
218 2 CALL INTERPROCESSOR (100);
00040610
00040620
00040630
00040640
219 2 CALL NEWFLOW ('LOCAL PROJECTION');
220 2 PGS = CEIL (20 * RELATION_SIZE / PAGESIZE)
+ CEIL (10 * CLASS_SIZE / PAGESIZE)
+ CEIL (20 * CLASS_SIZE / PAGESIZE);
221 2 CALL NEWSTEP ('CENTRAL PROCESSOR', 20000 + 100 * (CLASS_SIZE
+ RELATION_SIZE), 1);
222 2 CALL LOCALBLOCK (PGS * PAGESIZE);
00040650
00040660
00040670
00040680
00040690
00040691
00040700
223 2 CALL NEWFLOW ('REMOTE PROJECTION');
224 2 CALL INTERPROCESSOR (100);
225 2 CALL NEWSTEP ('CENTRAL PROCESSOR', 20000, 1);
226 2 CALL NEWSTEP ('REMOTE PROCESSOR', 20000 + 100 * (CLASS_SIZE
+ RELATION_SIZE), 1);
227 2 CALL REMOTEBLOCK (PGS * PAGESIZE);
00040710
00040720
00040730
00040740
00040741
00040750
228 2 CALL NEWFLOW ('LOCAL OUTPUT');
229 2 PGS = CEIL (20 * CLASS_SIZE / PAGESIZE);
230 2 CALL LOCALBLOCK (PGS * PAGESIZE);
00040760
00040770
00040780

```

```
SYMT LEVEL NEST
231 2 CALL NEWSTEP ('CENTRAL PROCESSOR', 2E4 + 100*CLASS_SIZE, 1); 00040790
232 2 CALL NEWFLW ('REMOTE OUTPLY'); 00040800
233 2 CALL NEWSTEP ('CENTRAL PROCESSOR', 2E4 + 100*CLASS_SIZE, 1); 00040810
234 2 CALL NEWSTEP ('REMOTE PROCESSOR', 2E4 + 100*CLASS_SIZE, 1); 00040820
235 2 CALL REMTEBLOCK (PGS * PAGESIZE); 00040830
236 2 CALL INTERPROCESSOR (20 * CLASS_SIZE); 00040840
237 2 CALL LOCALBLOCK (PGS * PAGESIZE); 00040850
```

STMT LEVEL NEST

/* THE TRANSACTIONS */

```

238 2 CALL NENTXN (*LOCN*, LGRATE);
239 2 CALL USEFLOW (*LOAD LCCFL PAGE*, 5);
00040870
00040880

240 2 CALL NENTXN (*CHANGE VERSION*, CVRATE);
241 2 CALL USEFLOW (*LOAD LCCFL PAGE*, 3);
00040890
00040900

242 2 CALL NENTXN (*SIMPLE QUERY*, SCRATE);
243 2 CALL USEFLOW (*PARSE LCCAL*, 1);
244 2 CALL USEFLOW (*LOCAL RECCD*, 6);
00040910
00040920
00040930

245 2 CALL NENTXN (*SIMPLE SENTENCE*, SSRATE);
246 2 CALL USEFLOW (*PARSE LCCAL*, 1+LATERAL);
247 2 CALL USEFLOW (*LOCAL PROJECTION*, 1+LATERAL+BASED);
248 2 CALL USEFLOW (*LOCAL OUTPUT*, 1);
00040950
00040960
00040970
00040980

249 2 CALL NENTXN (*COMPLEX QUERY*, MISTRATE);
250 2 PROJ = 0.5 * SENTLEN / (1 - PRECEFF);
251 2 CALL USEFLOW (*PARSE LOCAL*, 1 + PROJ*LATERAL);
252 2 CALL USEFLOW (*LOCAL PROJECTION*, PROJ * (1+PROJ*LATERAL+BASED));
253 2 CALL USEFLOW (*LOCAL OUTPUT*, 1);
00041020
00041030
00041040
00041050
00041060

```

/* REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 */

STMT LEVEL NEST

```

254 2 NENODE: PROCEDURE (NNAME, PGSIZE, RATE, USERS, SERVERS, QTYPE); 00050000
255 3 DECLARE
      NNAME CHARACTER (32),
      QTYPE BIN FIXED (31),
      (PGSIZE, RATE, USERS, SERVERS) BINARY FLOAT;
      #NODES = #NODES + 1;
      #NODE = ADDR (NODE(#NODES));
      THISNODE.NNAME = NNAME;
      THISNODE.PAGESIZE = PGSIZE;
      THISNODE.PAGERATE = RATE;
      THISNODE.USERS = USERS;
      THISNODE.SERVERS = SERVERS;
      THISNODE.QUEUETYPE = QTYPE;
00050010
00050020
00050030
00050040
00050050
00050060
00050070
00050080
00050090
00050100
00050110
00050120
264 3 ENC NENODE; 00050130

```

STMT LEVEL NEST

```
265 2 NEWFLCN: PROCEDURE (FNAME); 00050140
266 3 DECLARE FNAME CHAR(32); 00050150
267 3 #FLOWS = #FLOWS + 1; 00050160
268 3 PELOW = ADDR (FLOW(#FLOWS)); 00050170
269 3 THISFLCN.NAME = FNAME; 00050180
270 3 THISFLOW.FIRSTSTEP = #STEP+1; 00050190
271 3 END NEWFLCN; 00050200
```

STMT LEVEL NEST

```

272 2 NEWSTEP: PROCEDURE (NMODE, SIZE, TIMES); 00050210
273 3 DECLARE 00050220
      1 BINARY FIXED (31), 00050230
      NMODE CHAR (32), 00050240
      (SIZE, TIMES) BINARY FLDAT; 00050250
274 3 #STEP = #STEP + 1; 00050260
275 3 PSTEP = ADDR (STEP(#STEP)); 00050270
      /* LINEAR SEARCH FOR NMODE NAME */ 00050280
      DO I = 1 TO #NODES; 00050290
      IF NMODE.NAME(I) = NMODE THEN DO; 00050300
      THISSTEP.NODE = ADDR(NMODE(I)); 00050310
      FLCW.LASTSTEP(#FLOWS) = #STEP; 00050320
      THISSTEP.CATASIZE = SIZE; 00050330
      THISSTEP.MULTIPLIER = TIMES; 00050340
      RETURN; 00050350
      END; 00050360
      /* ELSE FALL THROUGH LCCP AND TRY NEXT NODE */ 00050370
      END; 00050380
      /* NO NODE FOUND */ 00050390
      #STEP = #STEP - 1; 00050400
      PUT SKIP EDIT ('INVALID NODE NAME ', NMODE, ' - STEP OMITTED') 00050410
      (A,A,A); 00050420
285 3 1 END NEWSTEP; 00050430
286 3 /* NO NODE FOUND */
287 3 #STEP = #STEP - 1;
      PUT SKIP EDIT ('INVALID NODE NAME ', NMODE, ' - STEP OMITTED')
      (A,A,A);
288 3 END NEWSTEP;

```

```

STMT LEVEL NEST
289 2 COPYFLW: PROCEDURE (NFLCW): 00050440
290 3 DECLARE NFLW CHAR(32), (I,J) BIN FIXED (31): 00050450
291 3 DO I = 1 TO NFLWS: 00050460
292 3 IF FLW.NMHE(I) = NFLW THEN DC: 00050470
293 3 DO J = FLOW(I).FIRSTSTEP TO FLCW(I).LASTSTEP: 00050480
294 3 #STEP = #STEP + 1: 00050490
295 3 STEPI#STEP) = STEF(J): 00050500
296 3 END: 00050510
297 3 RETURN: 00050520
298 3 END: 00050530
299 3 END: 00050540
300 3
301 3 PUT SKIP EDIT ('FLOW ',NFLCW,' NOT FOUND; NOT COPIED')(A,A,A): 00050550
302 3 END COPYFLOW: 00050560

```

STMT LEVEL NEST

```

303 2 NEWTXN: PROCEDURE (TNAME, WEIGHT); 00050570
304 3 DECLARE CHARACTER (32)*
      TNAME BINARY FLOAT; 00050580
      WEIGHT 00050590
      00050600
305 3 #TXNS = #TXNS + 1; 00050610
306 3 PTXN = ACDR (TRANSACTION(#TXNS)); 00050620
307 3 THISTXN.NAME = TNAME; 00050630
308 3 THISTXN.WEIGHT = WEIGHT; 00050640
309 3 THISTXN.FIRSTFLOW = #TXNFLW + 1; 00050650
310 3 END NEWTXN; 00050660

```


/* REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 */

STMT LEVEL NEST

```

311 2 USEFLW: PROCEDURE (WHICFLC, REP);
312 3 DECLARE
    WHICFLC CHARACTER(32),
    IFL BINARY FIXED (31),
    REP BINARY FLOAT;
313 3 #TXNFLOW = #TXNFLOW + 1;
314 3 #TXNFLOW = ADDR (TXNFLOW(#TXNFLOW));
315 3 DO IFL = 1 TO #FLOWS;
316 3 IF WHICFLC = FLOW.NAME(IFL) THEN DO;
318 3 THISTXNFLOW.FLOW = ADDR(FLOW(IFL));
319 3 THISTXNFLOW.REPETITIONS = REP;
320 3 TRANSACTION.LASTFLOW(#TXNS) = #TXNFLOW;
321 3 RETURN;
322 3 END;
323 3 1
/* NO FLOW FOUND */
324 3 #TXNFLOW = #TXNFLOW - 1;
/* GIVE UP ON THIS ONE AND TRY ANOTHER */
325 3 PUT SKIP EDIT ('UNKNOWN FLOW NAME ', WHICFLC, ' OMITTED')
    (A,A,A);
326 3 END USEFLW;
/* FINAL CODE FOR PROCEDURE *CONFIG* TO GENERATE REPORTS */
327 2 CALL PERFORM_CALCULATIONS;
00050670
00050680
00050690
00050700
00050710
00050720
00050730
00050740
00050750
00050760
00050770
00050780
00050790
00050800
00050810
00050820
00050830
00050840
00050850
00050860
00050870
00050880
00050890

```

```
STMT LEVEL NEST
328 2 PERFORM_CALCULATCN: PROCEDURE: 00050900
329 3 CALL CALCULATE_STEP_LOAD: 00050910
330 3 CALL CALCULATE_TXN_LOAD: 00050920
331 3 CALL CALCULATE_NODE_RHC: 00050930
332 3 CALL CALCULATE_STEP_TIMES: 00050940
333 3 CALL CALCULATE_FLOW_TIMES: 00050950
334 3 CALL CALCULATE_TXNFLOW_TIMES: 00050960
335 3 CALL CALCULATE_TXN_TIMES: 00050970
336 3 CALL CALCULATE_NODE_TIMES: 00050980
```

/* REL CONFIGURATION MCDEL - PL/1 IMPLEMENTATION VERSION 4.0 */

STAT LEVEL NEST

00050990

337 3 CALCULATE_STEP_LOAD: PROCEDURE:

00051000

338 4 DECLARE IS BINARY FIXED (31):

00051010
00051020
00051030
00051040
00051050
00051060
00051070

339 4 DO IS = 1 TO #STEP;
340 4 PSTEP = ADDR(STEP(IS));
341 4 FNODE = THISSTEP-NODE;
342 4 THISSTEP-PAGES = CEIL((THISSTEP-CATASIZE/THISNODE-PAGESIZE)
* THISSTEP-MULTIPLIER;
343 4 THISSTEP-LOAD = THISSTEP-PAGES * THISNODE-USERS;
344 4 END;

00051080

345 4 END CALCULATE_STEP_LOAD;

```

SYMT LEVEL NEST
346 3      CALCULATE_TXN_LOAD: PROCEDURE:          00051090
347 4      DECLARE (IT, ITF, IS) BINARY FIXED (2);  00051100
348 4      NODE_LOAD (*,*) = 0;                    00051110
349 4      DO IT = 1 TO #TXNS;
350 4      PTXN = ADDR(TRANSACTION(IT));
351 4      DO ITF = THISXN-FIRSTFLW TC THISXN-LASTFLOW;
352 4      PIXNFLOW = ADDR(TXNFCN(ITF));
353 4      PFLW = THISXNFLOW.FLOW;
354 4      DO IS = THISFLOW-FIRSTSTEP TC THISFLOW-LASTSTEP;
355 4      PSTEP = ADDR(STEP(IS));
356 4      PNODE = THISSTEP.ACCE;
357 4      THISNODE_LOAD(IT) = THISNODE_LOAD(IT)
      + ARRIVAL_RATE
      * THISXN.WEIGHT
      * THISXNFCN.REPETITIONS
      * THISSTEP.ICAD;
358 4      EAD;
359 4      END;
360 4      END;
361 4      END CALCULATE_TXN_LOAD;          00051280

```

/* REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 */

```

STRT LEVEL NEST
362 3      CALCULATE_NODE_RHG: PROCEDURE;
                                00051290
363 4      DECLARE
                                00051300
                                (IN, IT) BINARY FIXED (31),
                                00051310
                                NETLCAD BINARY FLOAT;
                                00051320
364 4      DO IN = 1 TO #NODES;
                                00051330
365 4 1      PNODE = ADDR(NODELIN);
                                00051340
366 4 1      THISNODE-CAPACITY = THISNODE-PAGEFATE * THISNODE-SERVERS;
                                00051350
367 4 1      NETLOAD = 0;
                                00051360
368 4 1      CC IT = 1 TO #TXNS;
                                00051370
369 4 2      NETLOAD = NETLOAD + THISNODE-LOAD(IT);
                                00051380
370 4 2      ENDS;
                                00051390
371 4 1      THISNODE-UTILIZATION = NETLCAD / THISNODE-CAPACITY;
                                00051400
372 4 1      IF THISNODE-UTILIZATION >= 1 THEN DO;
                                00051410
373 4 2      THISNODE-MEANFACTOR = 1E6;
                                00051420
374 4 2      THISNODE-SIGMAFACTOR = 0;
                                00051430
375 4 2      ENDS;
                                00051440
376 4 2      ELSE IF THISNODE-USERS = 1
                                00051450
377 4 1      THEN DO; THISNODE-MEANFACTOR=1; THISNODE-SIGMAFACTOR=0; ENDS;
                                00051460
378 4 1      ELSE CALL CALCULATE_QUEUEING;
                                00051470
379 4 1      /* ENDF *
                                00051480
380 4 1      ENDS;
                                00051490
381 4 1      ENDS;
                                00051500
382 4 1      ENDS;
                                00051510
383 4 1      ENDS;
                                00051520
384 4 1      ENDS;
                                00051530
385 5      DECLARE (M,P,B) BINARY FLOAT, ALLBUSY RETURNS (BINARY FLOAT);
                                00051540
386 5      M = THISNODE-SERVERS;
                                00051550
387 5      P = THISNODE-UTILIZATION;
                                00051560
388 5      IF M = 1 | THISNODE-QUEUEUETYPE = 1 | THISNODE-QUEUEUETYPE = 2
                                00051570
389 5      THEN DO;
                                00051580
390 5 1      THISNODE-MEANFACTOR = 1/(1-P);
                                00051590
391 5 1      THISNODE-SIGMAFACTOR = 1/(1-P);
                                00051600
392 5 1      ENDS;
                                00051610
393 5 1      ELSE DO;
                                00051620
394 5 1      B = ALLBUSY(M, P);
                                00051630
395 5 1      THISNODE-MEANFACTOR = 1 + B/(P*(1-P));
                                00051640
396 5 1      THISNODE-SIGMAFACTOR = SQRT(E*(2-B)+(M*(1-P))**2)/(M*(1-P));
                                00051650
397 5 1      ENDS;
                                00051660
398 5 1      /* ENDF */
                                00051670

```

STMT LEVEL NEST

```

398 5 ALLBUSY: PROCEDURE (M, P) RETURNS (BIN FLOAT):
399 6 DECLARE
      (P, X, Y, SUM, TERM) BINARY FLCLT,
      N BINARY FIXED (31);
400 6 TERM, SUM = 1;
401 6 DO N = 1 TO M-1;
402 6   TERM = TERM * M * P / N;
403 6   SUM = SUM + TERM;
404 6   END;
405 6 X = SUM;
406 6 Y = SUM + TERM*P/N;
407 6 RETURN ( (1-X/Y)/(1-P*(X/Y)) );
408 6 END ALLBUSY;
409 5 END CALCULATE_QUEUEING;
410 4 END CALCULATE_NODE_RHO;

```

```

STMT LEVEL NEST
423 3 CALCULATE_FLOW_TIMES: PROCEDURE; 00051950
424 4 DECLARE (IFL, IS) BINARY FIXED (31); 00051960
425 4 DO IFL = 1 TO #FLOWS; 00051970
426 4 PFLOW = ADDR(FLOW(IFL)); 00051980
427 4 THISFLOW.TIME, THISFLOW.SIGMA = OF 00051990
428 4 DO IS = THISFLOW.FIRSTSTEP TO THISFLOW.LASTSTEP; 00052000
429 4 PSTEP = ADDR(STEP(IS)); 00052010
430 4 THISFLOW.TIME = THISFLOW.TIME + THISSTEP.TIME; 00052020
431 4 THISFLOW.SIGMA = THISFLOW.SIGMA + THISSTEP.SIGMA**2; 00052030
432 4 END; 00052040
433 4 THISFLOW.SIGMA = SORT(THISFLOW.SIGMA); 00052050
434 4 END; 00052060
435 4 END CALCULATE_FLOW_TIMES; 00052070

```

```

STMT LEVEL NEST
411 3 CALCULATE_STEP_TIMES: PROCEDURE: 00051800
412 4 DECLARE
    XT BINARY FLOAT,
    IS BINARY FIXED (31);
    00051810
    00051820
    00051830
413 4 DC IS = 1 TO #STEP;
414 4 PSTEP = ADDR(STEP(1));
415 4 FNODE = THISSTEP.NODE;
416 4 IF THISNODE.QUEUE TYPE = 1
417 4 THEN XT = CEIL (THISSTEP.PAGES / THISNODE.SERVERS)
    / THISNODE.PAGERATE;
    00051840
    00051850
    00051860
    00051870
    00051880
    00051890
418 4 ELSE XT = THISSTEP.PAGES / THISNODE.PAGERATE;
419 4 THISSTEP.TIME = THISNODE.MEANFACTOR * XT;
420 4 THISSTEP.SIGMA = THISNODE.SIGMAFACTOR * XT;
421 4 END;
    00051900
    00051910
    00051920
    00051930
422 4 END CALCULATE_STEP_TIMES;
    00051940
    
```



```
STMT LEVEL NEST
436 3 CALCULATE_TXNFLOW_TIMES: PRCCEDLRE: 0005209C
437 4 DECLARE ITF BINARY FIXED (31): 00052090
438 4 DO ITF = 1 TO #TXNFLOW; 00052100
439 4 PIXNFLOW = ADDR(TXNFLOW(ITF)); 00052110
440 4 PFLCH = THISXNFLOW.FLCH; 00052120
441 4 THISXNFLOW.TIME = THISXNFLOW.REPETITIONS * THISFLOW.TIME; 00052130
442 4 THISXNFLOW.SIGMA = SQRT(THISXNFLOW.REPETITIONS) 00052140
443 4 * THISFLCH.SIGMA; 00052150
444 4 END; 00052160
444 4 END CALCULATE_TXNFLOW_TIMES: 00052170
```

STMT LEVEL NEST

```

445 3      CALCULATE_TXN_TIMES: PROCEDURE;          00052180
446 4      DECLARE (IT, ITF) BINARY FIXED (31);    00052190
447 4      DO IT = 1 TO #TXNS;
448 4          PTXN = ADDR(TRANSACTION(IT));
449 4          THISXN.TIME, THISXN.SIGMA = 0;
450 4          DO ITF = THISXN-FIRSTFCB TO THISXN-LASTFLOW;
451 4              PTXNLOW = ADDR(PTXNFCB(ITF));
452 4              THISXN.TIME = THISXN.TIME + THISXNLOW.TIME;
453 4              THISXN.SIGMA = THISXN.SIGMA + THISXNLOW.SIGMA**2;
454 4          END;
455 4          THISXN.SIGMA = SQRT(THISXN.SIGMA);
456 4          END;
457 4      END CALCULATE_TXN_TIMES;          00052300

```

```

STMT LEVEL NEST
458 3      CALCULATE NODE_TIMES: PROCEDURE;
                                00052310
459 4      DECLARE (IT, IIF, IS) BIN FIXED (31), X REAL;
                                00052320
460 4      MODEL*.TIME = 0;
461 4      WEIGHTED_MEAN_TIME = 0;
                                00052330
                                00052340
462 4      DO IT = 1 TO #TXNS;
463 4          ITXN = ADDR(TRANSACTION(IT));
464 4          DO IIF = THISXN.FIRSTFCM TO THISXN.LASTFLOW;
465 4              ITXNFLOW = ADDR (ITXNFLOW(IIF));
466 4              PELOW = THISXNFLOW.FLOW;
467 4              DO IS = THISFLOW.FIRSTSTEP TO THISFLOW.LASTSTEP;
468 4                  PSTEP = ADDR(STEP(IS));
469 4                  PNODE = THISSTEP.PNODE;
470 4                  X = THISSTEP.TIME + ITXNFLOW.REPETITIONS
                                * THISXN.WEIGHT;
471 4                  THISNODE.TIME = THISNODE.TIME + X;
472 4                  WEIGHTED_MEAN_TIME = THISNODE.TIME + X;
473 4                  END;
474 4              END;
475 4          END;
476 4          END CALCULATE_NODE_TIMES;
                                00052350
                                00052360
                                00052370
                                00052380
                                00052390
                                00052400
                                00052410
                                00052420
                                00052430
                                00052440
                                00052450
                                00052460
                                00052470
                                00052480
                                00052490
                                00052500
477 3      END PERFORM_CALCULATIONS;
                                00052510
478 2      END CONFIG;
                                00052520

```

STMT LEVEL NEST			
479	1	REPORT: PROCEDURE:	00052530
480	2	CALL OUTPUT_NODES:	00052540
481	2	CALL OUTPUT_FLOWS:	00052550
482	2	CALL OUTPUT_TXNS:	00052560

```

STMT LEVEL NEST
483 2 OUTPUT_NODES: PROCEDURE: 00052570
484 3 DECLARE (IN, IT) BINARY FIXED (31); 00052580
485 3 SIGNAL ENDPAGE (SYSPRINT);
486 3 PUT EDIT ('NODE', PAGE SIZE PAGE RATE USERS SERVERS',
487 3 , CAPACITY UTIL MEAN/SO SIGMA/SO WCTD TIME*) (R(FORM1));
488 3 PUT SKIP EDIT ('-----') (R(FORM1));
489 3 PUT SKIP(2);
490 3 FCRM1: FORMAT (A(32),A(40),A(50));
491 3 PUT EDIT (NODE-NAME(IN), NODE-PAGESIZE(IN), NODE-PAGERATE(IN),
492 3 NODE-USERS(IN), NODE-SERVERS(IN),
493 3 NODE-CAPACITY(IN), NODE-UTILIZATION(IN),
494 3 NODE-MEANFACTOR(IN), NODE-SIGMAFACTOR(IN), NODE-TIME(IN)
495 3 DO IN = 1 TO #NODES)
496 3 (SKIP(2),A(32),A(40),A(50));
497 3 SIGNAL ENDPAGE(SYSPRINT);
498 3 PUT EDIT ('NODE', TRANSACTION', PERCENT OF CAPACITY') (R(FORM2));
499 3 PUT SKIP EDIT ('-----') (R(FORM2));
500 3 FORM2: FORMAT (I(3),X(11));
501 3 DO IN = 1 TO #NODES;
502 3 DO IT = 1 TO #TXNS;
503 3 PUT SKIP EDIT (NODE-NAME(IN), TRANSACTION-NAME(IT),
504 3 100*NODE(IN)-LOAD(IT)/NODE(IN)-CAPACITY
505 3 ((2),A(32),X(11)),F(12,4));
506 3 ENC;
507 3 PUT SKIP;
508 3 END;
509 3 END OUTPUT_NODES: 00052860

```

```

STMT LEVEL NEST
503 2 OUTPUT_FLCS: PROCEDURE; 00052870
504 3 DECLARE
      (IFL, IS) BINARY FIXED (31),
      CHAR32 CHARACTER (32) BASEC (PCHAR32);
      00052880
      00052890
      00052900
505 3 SIGNAL ENDPAGE (SYSPRINT);
      00052910
506 3 PUT EDIT ('FLOW', *NODE*, * CATA SIZE PAGES TIME SIGMA*')
      (R(FORM3));
      00052920
      00052930
507 3 PUT SKIP EDIT ('-----', *-----',
      '-----') (R(FORM3));
      00052940
      00052950
      00052960
508 3 FORM3: FORMAT ((2)(A(32),X(1)),A(40));
      00052970
509 3 DO IFL = 1 TO #FLOWS;
      00052980
510 3 PUT EDIT (FLOW-NAME(IFL)) (SKIP(3),A(32));
      00052990
511 3 DO IS = FLOW(IFL).FIRSTSTEP TO FLOW(IFL).LASTSTEP;
      00053000
512 3 PCHAR32 = STEP-NODE(IS); /* FL/I-F RESTRICTION */
      00053010
513 3 PUT EDIT (CHAR32,
      STEP-DATASIZE(IS), STEP-PAGES(IS),
      STEP-TIME(IS), STEP-SIGMA(IS))
      (SKIP(0),X(33),A(32),X(1)),(2)(F(10)),(2)(F(10,3)));
      00053020
      00053030
      00053040
      00053050
      00053060
514 3 PUT SKIP (2);
      00053070
515 3 END;
      00053080
516 3 PUT EDIT ('TOTAL FOR FLOW',FLOW-TIME(IFL), FLOW-SIGMA(IFL))
      (SKIP(0),X(33),A(32),X(21)),(2)(F(10,3));
      00053090
517 3 END;
      00053100
518 3 END OUTPUT_FLOWS;

```

STMT LEVEL NEST

```

519 2 OUTPUT_TXNS: PROCEDURE;                                00053110
520 3 DECLARE
521 3 (IT, ITF) BINARY FIXED (31),
522 3 CHAR32 CHARACTER (32) BASEC (PCHAR32);
523 3 SIGNAL ENDPAGE (SYSPRINT);
524 3 PUT EDIT (TRANSACTION, 'WEIGHT', 'FLOW',
525 3 REPS, TIME, SIGMA) (R(FORM4));
526 3 PUT SKIP EDIT ('-----', '-----', '-----',
527 3 '-----', '-----', '-----') (R(FORM4));
528 3 FORM4: FORMAT (A(32),X(5),A(6),X(11),A(32),X(1),A(30));
529 3 CO IT = 1 TO #TXNS;
530 3 PUT EDIT (TRANSACTION.NAME(IT), TRANSACTION.WEIGHT(IT),
531 3 (SKIP(3),A(32),X(1),X(10,6)));
532 3 DO ITF = TRANSACTION.FIRSTFLOW(IT) TO TRANSACTION.LASTFLOW(IT);
533 3 PCHAR32 = TXNFLOW(ITF).FLOW; /* PL/I-F RESTRICTION */
534 3 PUT EDIT (CHAR32,
535 3 TXNFLOW(ITF).REPEATITIONS, TXNFLOW(ITF).TIME,
536 3 TXNFLOW(ITF).SIGMA,
537 3 (SKIP(10),X(44),A(32),X(1),X(3)) (F(10,3)));
538 3 ENCL;
539 3 PUT SKIP (2);
540 3 ENCL;
541 3 PUT EDIT (*TOTAL FOR TRANSACTIONS, TRANSACTION.TIME(IT),
542 3 TRANSACTION.SIGMA(IT)) (SKIP(10),X(44),A(32),X(11),X(2)) (F(10,3));
543 3 END;
544 3 END OUTPUT_TXNS;
545 2 END REPORT;
546 1 END_OF_JOB;
547 1 END FREQ;

```

A-35

APPENDIX B DETAIL OUTPUT SAMPLE

<u>Page</u>	<u>Report</u>
B-2	Node Summary
B-3	Node Utilization by Transaction
B-4	Flow Detail
B-6	Transaction Detail

NODE	PAGE SIZE	PAGE RATE	LSERS	SERVERS	CAPACITY	UTIL	MEAN/SD	SIGMA/SD	WGTD TIME
CENTRAL PROCESSOR	1	1000000	100	1	1000000	0.5553	2.2486	2.2486	1.2486
LOCAL DISK ARM	1	33	1	1	33	0.0000	1.0000	0.0000	0.0000
LOCAL DISK CTRL	2000	66	1	1	86	0.0000	1.0000	0.0000	0.0000
ETHERNET	0	1000000	100	1	1000000	0.0000	1.0000	1.0000	0.0000
REMOTE PROCESSOR	1	1000000	100	17	17000000	0.0000	1.0000	1.0000	0.0000
REMOTE DISK	1	33	6	6	198	0.0000	1.0000	1.0000	0.0000
REMOTE CTRL	2000	66	6	6	514	0.0000	1.0000	1.0000	0.0000

NCODE	TRANSACTION	PERCENT OF CAPACITY
CENTRAL PROCESSOR	LOGON	0.0100
CENTRAL PROCESSOR	CHANGE VERSION	0.0180
CENTRAL PROCESSOR	SIMPLE QUERY	1.6011
CENTRAL PROCESSOR	SIMPLE SENTENCE	11.2412
CENTRAL PROCESSOR	COMPLEX QUERY	42.6576
LOCAL DISK ARM	LOGON	0.0000
LOCAL DISK ARM	CHANGE VERSION	0.0000
LOCAL DISK ARM	SIMPLE QUERY	0.0000
LOCAL DISK ARM	SIMPLE SENTENCE	0.0000
LOCAL DISK ARM	COMPLEX QUERY	0.0000
LOCAL DISK CTLR	LOGON	0.0000
LOCAL DISK CTLR	CHANGE VERSION	0.0000
LOCAL DISK CTLR	SIMPLE QUERY	0.0000
LOCAL DISK CTLR	SIMPLE SENTENCE	0.0000
LOCAL DISK CTLR	COMPLEX QUERY	0.0000
ETHERNET	LOGON	0.0000
ETHERNET	CHANGE VERSION	0.0000
ETHERNET	SIMPLE QUERY	0.0000
ETHERNET	SIMPLE SENTENCE	0.0000
ETHERNET	COMPLEX QUERY	0.0000
REMOTE PROCESSOR	LOGON	0.0000
REMOTE PROCESSOR	CHANGE VERSION	0.0000
REMOTE PROCESSOR	SIMPLE QUERY	0.0000
REMOTE PROCESSOR	SIMPLE SENTENCE	0.0000
REMOTE PROCESSOR	COMPLEX QUERY	0.0000
REMOTE DISK	LOGON	0.0000
REMOTE DISK	CHANGE VERSION	0.0000
REMOTE DISK	SIMPLE QUERY	0.0000
REMOTE DISK	SIMPLE SENTENCE	0.0000
REMOTE DISK	COMPLEX QUERY	0.0000
REMOTE CTLR	LOGON	0.0000
REMOTE CTLR	CHANGE VERSION	0.0000
REMOTE CTLR	SIMPLE QUERY	0.0000
REMOTE CTLR	SIMPLE SENTENCE	0.0000
REMOTE CTLR	COMPLEX QUERY	0.0000

FLOW	MODE	DATA SIZE	PAGES	TIME	SIGMA
LOAD LOCAL PAGE	CENTRAL PROCESSOR	2000	2000	0.004	0.004
	TOTAL FOR FLOW			0.004	0.004
LOAD REMOTE PAGE	REMOTE PROCESSOR	2000	2000	0.002	0.002
	CENTRAL PROCESSOR	10000	10000	0.022	0.022
	ETHERNET	521	16680	0.017	0.017
	REMOTE PROCESSOR	10000	10000	0.010	0.010
	TOTAL FOR FLOW			0.051	0.030
PARSE LOCAL	CENTRAL PROCESSOR	2000	2000	0.004	0.004
	CENTRAL PROCESSOR	36035	36035	0.081	0.081
	TOTAL FOR FLOW			0.086	0.081
PARSE REMOTE	CENTRAL PROCESSOR	4000	4000	0.009	0.009
	ETHERNET	121	970	0.001	0.001
	REMOTE PROCESSOR	4000	4000	0.004	0.004
	REMOTE PROCESSOR	2000	2000	0.002	0.002
	REMOTE PROCESSOR	36035	36035	0.036	0.036
	TOTAL FOR FLOW			0.052	0.037
LOCAL RECORD	CENTRAL PROCESSOR	2000	2000	0.004	0.004
	TOTAL FOR FLOW			0.004	0.004
REMOTE RECORD	CENTRAL PROCESSOR	4000	4000	0.009	0.009
	ETHERNET	121	970	0.001	0.001
	REMOTE PROCESSOR	4000	4000	0.004	0.004
	REMOTE PROCESSOR	2000	2000	0.002	0.002
	CENTRAL PROCESSOR	4000	4000	0.009	0.009
	TOTAL FOR FLOW			0.026	0.026
	ETHERNET	121	970	0.001	0.001

REMOTE PROCESSOR	4000	4000	0.004	0.004
TOTAL FOR FLCH			0.030	0.014
CENTRAL PROCESSOR	170000	170000	0.382	0.362
CENTRAL PROCESSOR	2000	2000	0.004	0.004
TOTAL FOR FLCH			0.387	0.382
CENTRAL PROCESSOR	4000	4000	0.009	0.009
ETHERNET	121	970	0.001	0.001
REMOTE PROCESSOR	4000	4000	0.004	0.004
CENTRAL PROCESSOR	20000	20000	0.045	0.045
REMOTE PROCESSOR	170000	170000	0.170	0.170
REMOTE PROCESSOR	2000	2000	0.002	0.002
TOTAL FOR FLCH			0.231	0.176
CENTRAL PROCESSOR	2000	2000	0.004	0.004
CENTRAL PROCESSOR	70000	70000	0.157	0.157
TOTAL FOR FLCH			0.162	0.157
CENTRAL PROCESSOR	70000	70000	0.157	0.157
REMOTE PROCESSOR	70000	70000	0.070	0.070
REMOTE PROCESSOR	2000	2000	0.002	0.002
CENTRAL PROCESSOR	42000	42000	0.094	0.094
ETHERNET	521	83400	0.083	0.083
REMOTE PROCESSOR	42000	42000	0.042	0.042
CENTRAL PROCESSOR	2000	2000	0.004	0.004
TOTAL FOR FLCH			0.454	0.218

LOCAL OUTPUT

REMOTE OUTPUT

TRANSACTION	WEIGHT FLOW	REPS	TIME	SIGMA
LOGCN	0.010000 LCAD LCCAL PAGE	5.000	0.022	0.010
	TCTAL FCP TRANSACTION		0.022	0.010
CHANGE VERSION	0.030000 LOAD LCCAL PAGE	3.000	0.013	0.008
	TCTAL FCP TRANSACTION		0.013	0.008
SIMPLE QUERY	0.320000 PARSE LCCAL	1.000	0.086	0.081
	LCCAL RECRD	6.000	0.027	0.011
	TCTAL FCP TRANSACTION		0.113	0.082
SIMPLE SENTENCE	0.320000 PARSE LCCAL	1.125	0.096	0.086
	LOCAL PROJECTION	1.375	0.532	0.448
	LCCAL CUTPUT	1.000	0.162	0.157
	TCTAL FCP TRANSACTION		0.790	0.483
COMPLEX QUERY	0.320000 PARSE LCCAL	1.500	0.128	0.099
	LCCAL PROJECTION	7.000	2.707	1.011
	LCCAL CUTPUT	1.000	0.162	0.157
	TCTAL FCP TRANSACTION			

APPENDIX C COST SUMMARIES OF THE CASES

<u>Page</u>	<u>Case</u>
C- 2	Paging Disk Centralized
C- 3	Paging Disk Clustered
C- 4	Paging Disk Smart Terminal
C- 5	Paging Bubble Centralized
C- 6	Paging Bubble Clustered
C- 7	Paging Bubble Smart Terminal
C- 8	Paging CCD Centralized
C- 9	Paging CCD Clustered
C-10	Paging CCD Smart Terminal
C-11	Paging EBAM Centralized
C-12	Paging EBAM Clustered
C-13	Paging EBAM Smart Terminal
C-14	Distributive Function Disk Centralized
C-15	Distributive Function Disk Clustered
C-16	Distributive Function Disk Smart Terminal
C-17	Logic Per Head Disk Centralized
C-18	Logic Per Head Disk Clustered
C-19	Logic Per Head Disk Smart Terminal
C-20	Logic Per Track Bubble Centralized
C-21	Logic Per Track Bubble Clustered
C-22	Logic Per Track Bubble Smart Terminal
C-23	Logic Per Track CCD Centralized
C-24	Logic Per Track CCD Clustered
C-25	Logic Per Track CCD Smart Terminal
C-26	Single Level Store EBAM Centralized
C-27	Single Level Store EBAM Clustered
C-28	Single Level Store EBAM Smart Terminal
C-29	Single Level Store RAM Centralized
C-30	Single Level Store RAM Clustered
C-31	Single Level Store RAM Smart Terminal
C-32	Logic Per Head Disk: $c(R1)/c(R2)$ Analysis
C-33	Distributed Function: $c(R1)/c(R2)$ Analysis

Cost Function: Paging Disk Centralized Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)		1	2	5	10	20	50	100	200	500
500	35,650	18,325	7,930	4,465	2,767	1,727	1,373	1,219	1,162	
1,000	35,650	18,325	7,930	4,535	2,767	1,727	1,419	1,242	1,179	
2,000	35,650	18,325	8,070	4,535	2,817	1,767	1,449	1,382	1,228	
5,000	35,650	18,475		8,155	4,577		2,281	1,994	100 SMALL	
10,000										T00 SMALL
20,000										T00 SMALL
50,000										T00 SMALL
100,000										T00 SMALL
200,000										T00 SMALL
500,000										T00 SMALL

Cost Function: Paging Disk Clustered Architecture

CONFIGURATION	1	2	5	10	20	50	100	200	500
500	2,151	2,151	2,151	2,191	2,151	2,191	2,191	2,191	2,191
1,000	2,151	2,191	2,151	2,191	2,151	2,191	2,191	2,151	2,191
2,000	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308
5,000	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308
10,000	TCC SMALL								
20,000	TCC SMALL								
50,000	TCC SMALL								
100,000	TCC SMALL								
200,000	TCC SMALL								
500,000	TCC SMALL								

Cost Function: Paging Disk Smart Terminal Architecture

CONFIGURATION	1	2	5	10	20	50	100	200	500
100	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550
1,000	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550
2,000	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550	5,550
5,000	6,250	6,250	6,250	6,250	6,250	6,250	6,250	6,250	6,250
10,000 TCO SMALL	6,250	6,250	6,250	6,250	6,250	6,250	6,250	6,250	6,250
20,000 TCC SMALL									
50,000 TCO SMALL									
100,000 TCO SMALL									
200,000 TCC SMALL									
500,000 TCO SMALL									

Cost Function: Paging Bubble Centralized Architecture

REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 OF 22-AUG-1979 - RUN DATE: 26 FEBRUARY 1980 - PAGE 12

CONFIGURATION COST VCF NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)

	1
500 TCC SMALL	
1,000 TCC SMALL	
2,000 TCC SMALL	
5,000 TCC SMALL	
10,000 TCC SMALL	
20,000 TCC SMALL	
50,000 TCC SMALL	
100,000 TCC SMALL	
200,000 TCC SMALL	
500,000 TCC SMALL	

Cost Function: Paging Bubble Clustered Architecture

- PAGE 12

REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 OF 22-ALG-1975 - RUN DATE: 26 FEBRUARY 1980

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUBES (ROW)

1

500 T00 SMALL
1,000 T00 SMALL
2,000 T00 SMALL
5,000 T00 SMALL
10,000 T00 SMALL
20,000 T00 SMALL
50,000 T00 SMALL
100,000 T00 SMALL
200,000 T00 SMALL
500,000 T00 SMALL

Cost Function: Paging Bubble Smart Terminal Architecture

REL CONFIGURATION MODEL - 61/1 IMPLEMENTATION VERSION 4.0 OF 22-AUG-1975 - RUN DATE: 26 FEBRUARY 1980 - PAGE 12

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)

1

500	T00	SMALL
1,000	T00	SMALL
2,000	T00	SMALL
5,000	T00	SMALL
10,000	T00	SMALL
20,000	T00	SMALL
50,000	T00	SMALL
100,000	T00	SMALL
200,000	T00	SMALL
500,000	T00	SMALL

Cost Function: Paging CCD Centralized Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	32,476	17,068	7,823	4,748	3,210	2,288	1,980	1,825	1,774
1,000	33,004	17,556	8,351	5,276	3,738	2,816	2,508	2,455	2,302
2,000	34,060	18,652	8,407	6,332	4,794	3,872	3,564	3,511	3,520
5,000	37,228	21,820	12,575	9,500	7,962	7,445	6,934	6,679	6,688
10,000	42,508	27,100	17,855	14,780	14,255	12,725	12,214	12,364	TOD SMALL
20,000	72,318	47,785	32,465	27,365	24,815	23,285	23,585	22,925	TOD SMALL
50,000	165,998	119,565	80,345	67,145	60,545	56,587	TOD SMALL		
100,000	238,758	172,758	133,158	119,945	113,345	TOD SMALL			
200,000	TOD SMALL								
500,000	TOD SMALL								

Cost Function: Paging CCD Clustered Architecture

	1	2	5	10	20	50	100	200	500
500	3,336	3,336	3,336	3,336	3,336	3,336	3,336	3,336	3,336
1,000	4,392	4,392	4,392	4,392	4,392	4,392	4,392	4,392	4,392
2,000	6,504	6,504	6,504	6,504	6,504	6,504	6,504	6,504	6,504
5,000	12,840	12,840	12,840	12,840	12,840	12,840	12,840	12,840	12,840
10,000	23,400	23,400	23,400	23,400	23,400	23,400	23,400	23,400	23,400
20,000	44,520	44,520	44,670	44,670	44,670	44,670	44,670	44,670	44,670
50,000	108,330	108,330	108,330	108,330	108,330	108,330	108,330	108,330	108,330
100,000	214,541	214,541	214,541	214,541	214,541	214,541	214,541	214,541	214,541
200,000	TOD SMALL								
500,000	TOD SMALL								

Cost Function: Paging CCD Smart Terminal Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20
500	16,576	11,601	8,696	7,661	TCC SMALL
1,000	TCC SMALL				
2,000	TCC SMALL				
5,000	TCC SMALL				
10,000	TCC SMALL				
20,000	TCC SMALL				
50,000	TCC SMALL				
100,000	TCC SMALL				
200,000	TCC SMALL				
500,000	TCC SMALL				

Cost Function: Paging EBAM Centralized Architecture

REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSILA 4.0 CF 22-AUG-1979 - RUN DATE: 22 FEBRUARY 1980 - PAGE 42

CONFIGURATION COST	VERSUS RELATION SIZE IN TUPLES (ROW)									
	1	2	5	10	20	50	100	200	500	
500	36,350	18,675	8,270	4,825	2,917	1,847	1,571	1,399	1,291	
1,000	36,350	19,175	8,270	4,835	2,917	1,847	1,571	1,433	1,419	
2,000	37,350	19,115	8,670	4,835	3,117	2,142	1,798	1,728	1,527	
5,000	39,350	22,175	9,470	5,910	3,855	2,732	2,456	2,183	1,999	
10,000	63,600	32,300	15,120	8,060	5,330	3,912	3,366	3,498	2,943	
20,000	71,600	36,300	18,320	12,360	8,280	5,732	4,716	4,038	TOD SMALL	
50,000	168,600	84,800	40,920	20,560	14,180	10,992	10,716	TOD SMALL		
100,000	200,600	132,800	53,720	38,160	25,980	20,432	TOD SMALL			
200,000	TOD SMALL									
500,000	TOD SMALL									

Cost Function: Paging EBAM Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)	RELATION SIZE IN TUPLES									
	1	2	5	10	20	50	100	200	500	
500	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308	
1,000	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	
2,000	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	
5,000	2,808	2,808	2,808	2,808	2,808	2,808	2,808	2,808	2,808	
10,000	3,475	3,475	3,475	3,475	3,475	3,475	3,475	3,475	3,475	
20,000	4,808	4,808	4,808	4,808	4,808	4,808	4,808	4,808	4,808	
50,000	12,558	12,958	12,958	12,958	12,958	12,958	12,958	12,958	12,958	
100,000	13,258	23,925	23,925	23,925	23,925	23,925	23,925	23,925	23,925	
200,000	24,525	24,525	45,858	45,858	45,858	45,858	45,858	45,858	45,858	
500,000	TOO SMALL									

Cost Function: Paging EBAM Smart Terminal Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNA)	VERSUS RELATION SIZE IN TUPLES (ROW)									
	1	2	5	10	20	50	100	200	500	
500	1,850	7,850	7,850	7,850	7,850	7,850	7,850	7,850	7,850	
1,000	6,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850	
2,000	8,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850	
5,000	10,850	10,850	10,850	10,850	10,850	10,850	10,850	10,850	10,850	
10,000	14,850	14,850	14,850	14,850	14,850	14,850	14,850	14,850	14,850	
20,000	22,850	22,850	22,850	22,850	22,850	22,850	22,850	22,850	22,850	
50,000	71,750	71,750	71,750	71,750	71,750	71,750	71,750	71,750	71,750	
100,000	137,550	137,550	137,550	137,550	137,550	137,550	137,550	137,550	137,550	
200,000	269,150	269,150	269,150	269,150	269,150	269,150	269,150	269,150	269,150	
500,000 TCC SMALL										

CONFIGURATION COST FOR NUMBER OF USEPS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW) PIRATE= 3.199955E-01:

Cost Function: Distributive Function Disk Centralized

	1	2	5	10	20	50	100	200	500
500	22,050	11,525	5,210	3,105	2,122	1,449	1,264	1,191	1,123
1,000	22,050	11,525	5,210	3,245	2,122	1,529	1,322	1,220	1,147
2,000	22,050	11,525	5,210	3,245	2,122	1,529	1,322	1,220	1,147
5,000	22,050	11,525	5,450	3,245	2,322	1,769	1,500	1,397	1,247
10,000	22,050	12,225	5,450	3,635	2,617	2,001	1,736	1,633	1,631
20,000	25,250	14,175	7,050	4,825	3,502	2,591	2,267	2,164	2,309
50,000	34,850	21,575	10,170	8,365	6,197	4,479	3,860	3,580	3,832
100,000	57,250	37,575	17,150	13,675	9,657	7,311	6,692	6,088	6,599
200,000	92,450	60,975	25,670	18,455	16,547	13,568	12,179	10,746	12,491
500,000	217,250	144,825	62,550	57,525	43,022	30,799	27,460	26,685	29,189

CONFIGURATION COST FOR NUMBER OF USEKS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW) MISRATE= 3.199999E-01

Cost Function: Distributive Function Disk Clustered Architecture

	1	2	5	10	20	50	100	200	500
500	19,100	10,050	4,620	4,620	4,620	4,258	4,077	4,077	4,040
1,000	19,100	10,050	4,620	4,620	4,620	4,258	4,077	4,077	4,040
2,000	19,100	10,050	4,620	4,620	4,620	4,258	4,077	4,077	4,040
5,000	19,100	10,050	4,900	4,900	4,900	4,510	4,315	4,315	4,276
10,000	19,100	10,750	4,900	4,900	4,900	4,510	4,315	4,315	4,276
20,000	25,500	14,650	6,460	6,460	6,460	5,914	5,641	5,641	5,586
50,000	38,300	22,450	9,580	9,580	9,580	8,722	8,293	8,293	8,207
100,000	60,700	36,100	15,040	15,040	15,040	13,636	12,934	12,934	12,793
200,000	108,700	65,350	26,740	26,740	26,740	24,166	22,879	22,879	22,621
500,000	259,100	157,000	63,400	63,400	63,400	57,160	54,040	54,040	53,416

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROWS) MISRATE= 3.199999E-01:

Cost Function: Distributive Function Disk Smart Terminal

	1	2	5	10	20	50	100	200	500
500	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100
1,000	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100
2,000	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100	18,100
5,000	18,100	18,100	19,500	19,500	19,500	19,500	19,500	19,500	19,500
10,000	18,100	19,500	19,500	19,500	19,500	19,500	19,500	19,500	19,500
20,000	21,300	23,400	23,400	23,400	23,400	23,400	23,400	23,400	23,400
50,000	34,100	39,000	39,000	39,000	39,000	39,000	39,000	39,000	39,000
100,000	53,300	62,400	62,400	62,400	62,400	62,400	62,400	62,400	62,400
200,000	98,100	117,000	117,000	117,000	117,000	117,000	117,000	117,000	117,000
500,000	213,300	257,400	257,400	257,400	257,400	257,400	257,400	257,400	257,400

Cost Function: Logic Per Head Disk Centralized

	1	2	5	10	20	50	100	200	500
CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)									
500	45,657	23,328	9,931	5,456	3,278	1,936	1,740	1,502	1,401
1,000	45,657	23,328	9,931	5,456	3,278	1,936	1,740	1,502	1,401
2,000	45,657	23,328	9,931	5,456	3,278	1,936	1,740	1,502	1,401
5,000	45,657	23,328	9,931	5,456	3,278	1,936	1,745	1,409	1,366
10,000	45,657	23,328	9,931	5,456	3,278	1,939	1,663	1,458	1,422
20,000	45,964	23,482	9,952	5,456	3,278	2,139	1,863	1,709	1,629
50,000	46,578	23,789	10,115	5,560	3,966	2,936	2,696	2,524	2,455
100,000	46,578	24,403	10,364	6,929	5,212	4,253	3,977	3,857	3,744
200,000	47,807	25,632	13,840	10,405	8,867	7,801	7,561	7,389	7,313
500,000	55,185	38,010	27,705	24,630	22,913	21,954	21,678	21,558	21,445

Cost Function: Logic Per Head Disk Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROWS)

	1	2	5	10	20	50	100	200	500
500	18,157	5,578	4,431	4,431	4,431	4,088	3,916	3,916	3,882
1,000	18,157	9,578	4,431	4,431	4,431	4,088	3,916	3,916	3,882
2,000	18,157	9,578	4,431	4,431	4,431	4,088	3,916	3,916	3,882
5,000	18,157	9,578	4,431	4,431	4,431	4,088	3,916	3,916	3,882
10,000	18,157	5,578	4,431	4,431	4,431	4,088	3,916	3,916	3,882
20,000	18,464	5,722	4,431	4,431	4,431	4,088	3,916	3,916	3,882
50,000	15,078	10,039	4,615	4,615	4,615	4,254	4,073	4,073	4,037
100,000	20,307	10,652	4,664	5,363	5,363	4,927	4,709	4,709	4,665
200,000	22,765	11,882	8,340	9,534	9,534	8,680	8,254	8,254	8,168
500,000	27,665	24,260	22,205	26,172	26,172	23,655	22,397	22,397	22,145

Cost Function: Logic Per Head Disk Smart Terminal Architecture

(output not available)

Cost Function: Logic Per Track Bubble Centralized Architecture

	1	2	5	10	20	50	100	200	500
500	22,472	17,036	7,654	4,567	3,025	2,008	1,711	1,581	1,774
1,000	33,072	17,636	8,134	5,059	3,522	2,402	2,114	1,975	2,392
2,000	34,355	18,897	9,119	6,044	4,506	3,190	2,922	2,762	3,629
5,000	37,041	21,420	11,186	8,013	6,475	7,837	7,608	7,488	6,080
10,000	42,333	26,466	15,223	11,951	10,414	7,916	7,766	7,646	11,058
20,000	52,916	36,558	22,256	19,828	18,290	14,217	14,224	14,104	21,014
50,000	74,082	56,741	39,442	35,581	34,043	52,025	52,976	51,597	40,926
100,000	116,414	97,107	71,735	67,087	65,550	102,435	104,644	102,008	80,751
200,000	201,078	177,839	136,321	130,100	128,563	203,256	207,981	202,828	160,401
500,000	370,407	355,032	511,252	508,178	506,640	404,896	414,656	404,469	319,755

Cost Function: Logic Per Track Bubble Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)	RELATION SIZE IN TUPLES (ROW)									
	1	2	5	10	20	50	100	200	500	
500	7,745	4,672	2,709	2,709	2,709	2,538	2,452	2,452	2,435	
1,000	8,345	5,272	3,189	3,189	3,189	2,970	2,860	2,860	2,838	
2,000	5,791	6,595	4,198	4,198	4,198	3,878	3,718	3,718	3,686	
5,000	12,683	9,241	6,216	6,216	6,216	5,694	5,434	5,434	5,381	
10,000	18,466	14,533	10,253	10,253	10,253	9,327	8,865	8,865	8,772	
20,000	30,032	25,116	16,326	16,326	16,326	16,593	15,727	15,727	15,554	
50,000	53,164	46,282	34,472	34,472	34,472	58,773	55,563	55,563	54,922	
100,000	95,428	88,614	66,765	66,765	66,765	115,485	109,124	109,124	107,852	
200,000	191,957	173,278	131,351	131,351	131,351	228,908	216,246	216,246	213,714	
500,000	439,929	374,664	518,865	518,865	518,865	467,079	441,185	441,185	436,007	

Cost Function: Logic Per Track Bubble Smart Terminal Architecture

REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 OF 22-AUG-1979 - RUN DATE: 25 MARCH 1980 - PAGE 60

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	6,745	6,745	6,745	6,745	6,745	6,745	6,745	6,745	6,745
1,000	7,345	7,345	7,345	7,345	7,345	7,345	7,345	7,345	7,345
2,000	8,545	8,545	8,791	8,791	8,791	8,791	8,791	8,791	8,791
5,000	11,191	11,683	11,683	11,683	11,683	11,683	11,683	11,683	11,683
10,000	16,483	16,483	17,466	17,466	17,466	17,466	17,466	17,466	17,466
20,000	27,066	27,066	29,032	29,032	29,032	29,032	29,032	29,032	29,032
50,000	52,164	52,164	52,164	52,164	52,164	52,164	52,164	52,164	52,164
100,000	58,428	58,428	58,428	58,428	58,428	58,428	58,428	58,428	58,428
200,000	190,957	190,957	190,957	190,957	190,957	190,957	190,957	190,957	190,957
500,000	376,014	376,014	376,014	376,014	376,014	376,014	376,014	376,014	376,014

Cost Function: Logic Per Track CCD Centralized Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	32,783	17,221	7,884	4,778	3,226	2,294	1,992	1,958	1,813
1,000	33,311	17,749	8,412	5,306	3,754	2,822	2,520	2,486	2,341
2,000	34,367	18,805	9,468	6,362	4,810	3,896	3,613	3,542	3,397
5,000	37,535	21,572	12,636	9,530	8,024	7,064	6,781	6,710	6,565
10,000	42,815	27,253	17,916	14,903	13,304	12,418	12,208	11,990	11,845
20,000	54,256	38,274	28,660	25,463	24,048	22,978	22,768	22,550	22,455
50,000	85,976	69,954	61,078	57,511	55,728	54,953	54,516	54,263	54,139
100,000	142,463	124,557	113,878	110,311	109,265	107,753	107,316	107,097	106,993
200,000	248,063	230,197	222,427	217,386	214,865	213,487	212,983	212,832	212,686
500,000	579,608	554,370	535,227	534,186	532,003	530,556	530,120	529,865	529,565

Cost Function: Logic Per Track CCD Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATICA SIZE IN TUPLES (ROW)	1	2	5	10	20	50	100	200	500
500	6,541	5,100	3,036	3,181	3,181	2,963	2,854	2,854	2,632
1,000	10,869	6,528	3,924	4,175	4,175	3,857	3,698	3,698	3,667
2,000	15,525	5,384	5,700	6,162	6,162	5,646	5,387	5,387	5,336
5,000	25,908	16,160	16,311	11,406	11,406	10,366	9,845	9,845	9,741
10,000	31,219	21,455	15,597	17,749	17,749	16,074	15,236	15,236	15,069
20,000	41,840	32,046	26,165	30,433	30,433	27,490	26,018	26,018	25,724
50,000	73,643	63,787	57,874	68,473	68,473	61,726	58,352	58,352	57,678
100,000	126,689	116,710	110,723	131,883	131,883	118,794	112,250	112,250	110,941
200,000	246,755	229,609	219,216	261,509	261,509	235,458	222,433	222,433	219,827

500,000 TOO SMALL

Cost Function: Logic Per Track CCD Smart Terminal Architecture

REL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSION 4.0 CF 22-AUG-1979 - RUN DATE: 15 FEBRUARY 1980 - PAGE 60

	1	2	5	10	20	50	100	200	500
500	6,641	6,641	6,641	6,641	6,641	7,541	7,541	7,541	7,541
1,000	8,069	8,069	8,069	8,069	8,069	8,069	8,069	8,069	9,869
2,000	10,925	10,925	14,525	14,525	14,525	14,525	14,525	14,525	14,525
5,000	24,908	24,908	24,908	24,908	24,908	24,908	24,908	24,908	24,908
10,000	30,188	30,219	30,219	30,219	30,219	30,219	30,219	30,219	30,219
20,000	40,779	40,779	40,840	40,840	40,840	40,840	40,840	40,840	40,840
50,000	72,643	72,643	72,643	72,643	72,643	72,643	72,643	72,643	72,643
100,000	125,689	125,689	125,689	125,689	125,689	125,689	125,689	125,689	125,689
200,000	231,781	231,781	231,781	231,781	231,781	231,781	231,781	231,781	231,781
500,000	549,564	549,564	549,564	549,564	549,564	549,564	549,564	549,564	549,564

Cost Function: Single Level Store EBAM Centralized Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	36,350	18,675	8,270	4,625	2,917	1,847	1,503	1,365	1,237
1,000	36,350	19,175	8,270	4,835	2,917	1,847	1,571	1,399	1,291
2,000	37,350	15,115	8,670	4,835	3,117	2,007	1,731	1,593	1,473
5,000	39,350	20,175	9,470	5,235	3,855	2,597	2,456	2,048	1,999
10,000	51,350	26,175	12,420	6,710	4,992	3,912	3,096	2,958	2,727
20,000	71,600	36,300	18,320	5,660	8,280	5,732	4,916	3,498	TOD SMALL
50,000	168,600	84,800	40,920	20,560	14,180	10,992	9,636	TOD SMALL	
100,000	200,600	132,800	53,720	38,160	25,980	18,272	TOD SMALL		
200,000	TOD SMALL								
500,000	TOD SMALL								

Cost Function: Single Level Store EBAM Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)	1	2	5	10	20	50	100	200	500
500	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308	2,308
1,000	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475
2,000	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475	2,475
5,000	2,808	2,808	2,808	2,808	2,808	2,808	2,808	2,808	2,808
10,000	3,475	3,475	3,475	3,475	3,475	3,475	3,475	3,475	3,475
20,000	4,808	4,808	4,808	4,808	4,808	4,808	4,808	4,808	4,808
50,000	7,475	7,475	7,475	7,475	7,475	7,475	7,475	7,475	7,475
100,000	12,808	12,808	12,808	12,808	12,808	12,808	12,808	12,808	12,808
200,000	23,475	23,475	23,475	23,475	23,475	23,475	23,475	23,475	23,475
500,000	87,475	87,475	87,475	87,475	87,475	87,475	87,475	87,475	87,475

Cost Function: Single Level Store EBAM Smart Terminal Architecture

CELL CONFIGURATION MODEL - PL/I IMPLEMENTATION VERSILA 4.C CF 22-AUG-1979 - RUN DATE: 26 FEBRUARY 1980 - PAGE 60

CELL CONFIGURATION COST FOR NUMBER OF USERS (COLUMN) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	7,650	7,850	7,850	7,650	7,850	7,850	7,850	7,850	7,850
1,000	8,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850	8,850
2,000	8,850	8,850	8,850	8,650	8,850	8,850	8,850	8,850	8,850
5,000	10,850	10,850	10,850	10,650	10,850	10,850	10,850	10,850	10,850
10,000	14,850	14,850	14,850	14,650	14,850	14,850	14,850	14,850	14,850
20,000	22,850	22,850	22,850	22,650	22,850	22,850	22,850	22,850	22,850
50,000	38,850	38,850	38,850	38,650	38,850	38,850	38,850	38,850	38,850
100,000	70,850	70,850	70,850	70,650	70,850	70,850	70,850	70,850	70,850
200,000	134,850	134,850	134,850	134,650	134,850	134,850	134,850	134,850	134,850
500,000	518,850	518,850	518,850	518,650	518,850	518,850	518,850	518,850	518,850

Cost Function: Single Level Store RAM Centralized Architecture

(output not available)

Cost Function: Single Level Store RAM Clustered Architecture

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
1,000 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
2,000 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
5,000	30,094	24,719	21,494	25,163	25,163	22,746	21,538	21,538	21,296
10,000	44,742	35,367	36,142	42,741	42,741	38,567	36,479	36,479	36,062
20,000	74,039	68,664	65,439	77,897	77,897	70,207	66,362	66,362	65,593
50,000	161,930	157,455	155,290	185,326	185,326	166,895	157,677	157,677	155,833
100,000	310,214	303,939	304,054	363,427	363,427	327,184	309,063	309,063	305,438
200,000	606,782	602,707	TOO SMALL	TLC SMALL	TLC SMALL				
500,000	TOO SMALL								

Cost Function: Single Level Store RAM Smart Terminal

CONFIGURATION COST FOR NUMBER OF USERS (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROWS)

	1	2	5	10	20	50	100	200	500
500 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
1,000 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
2,000 TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG	TOO BIG
5,000	14,576	14,576	14,576	14,576	14,576	14,576	14,576	14,576	14,576
10,000	20,826	20,826	20,826	20,826	20,826	20,826	20,826	20,826	20,826
20,000	33,326	33,326	33,326	33,326	33,326	33,326	33,326	33,326	33,326
50,000	70,826	70,826	70,826	70,826	70,826	70,826	70,826	70,826	70,826
100,000	135,126	135,126	135,126	135,126	135,126	135,126	135,126	135,126	135,126
200,000	263,726	263,726	263,726	263,726	263,726	263,726	263,726	263,726	263,726
500,000 TOO SMALL									

Logic Per Head Disk: $c(R_1) / c(R_2)$ Analysis

CONFIGURATION COST FOR RELATIVE CLASS SIZE (COLUMNS) VERSUS RELATIVE SIZE IN TUPLES (ROW)

	1	2	5	10	20	50	100	200	500
500	1,446	1,446	1,446	1,446	1,446	1,455	1,468	1,492	3,183
1,000	1,446	1,446	1,446	1,446	1,455	1,468	1,492	1,904	4,824
2,000	1,446	1,446	1,446	1,455	1,468	1,492	1,904	3,183	14,798
3,000	1,447	1,457	1,457	1,471	1,500	2,410	5,246	15,617	221,151
10,000	1,550	1,558	1,558	1,555	1,724	2,589	16,429	82,029	
20,000	1,777	1,770	1,866	1,922	2,311	5,806	85,225		
30,000	2,311	2,475	3,025	3,547	4,967	259,536		TOO SMALL	
100,000	4,048	4,548	5,634	12,000	70,325			TOO SMALL	
200,000	7,977	13,920	21,972	125,348				TOO SMALL	
300,000	52,234	170,914	745,880	TOO SMALL					

Distributed Function Architecture: $c(R_1) / c(R_2)$ Analysis

CONFIGURATION COST FOR RELATIVE CLASS SIZE (COLUMNS) VERSUS RELATION SIZE IN TUPLES (ROW) MISRATE = 3.199999E-01:

	1	2	5	10	20	50	100	200	500
500	1,212	1,212	1,212	1,212	1,212	1,252	1,252	1,252	1,352
1,000	1,252	1,252	1,252	1,252	1,252	1,252	1,252	1,352	1,562
2,000	1,252	1,252	1,752	1,252	1,252	1,252	1,352	1,457	1,945
5,000	1,379	1,379	1,379	1,379	1,342	1,493	1,721	2,017	3,064
10,000	1,607	1,607	1,607	1,607	1,607	1,835	2,245	3,064	5,386
20,000	1,835	1,835	1,949	1,545	2,063	2,473	3,292	4,702	11,330
50,000	2,861	2,861	2,861	3,089	3,385	4,774	6,868	12,125	TOD SMALL
100,000	4,571	4,571	4,571	4,981	6,028	8,806	14,090	22,090	TOD SMALL
200,000	7,649	7,649	8,401	9,334	11,314	17,063	26,083	TOD SMALL	
500,000	17,111	18,775	20,734	23,130	28,743	TCC SMALL			