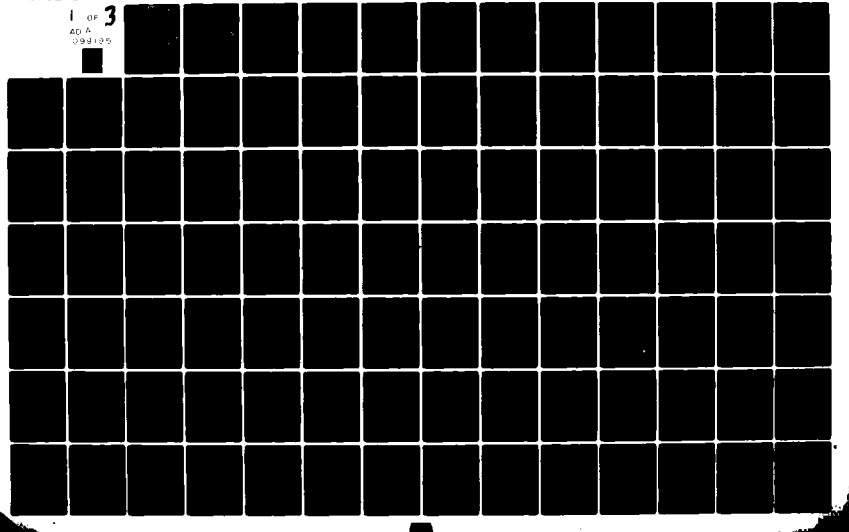
AD-A099 195 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
DECENTRALIZED SYSTEMS.(U)
DEC 80 E E BALKOVICH DAS660-79-C-0117
UNCLASSIFIED TR-CS-15-80 NL



LEVEL

COMPUTER SCIENCE TECHNICAL REPORT

Laboratory for Computer Science Research
The University of Connecticut

AD A099195

DTIC FILE COPY

DTIC
ELECTE
MAY 20 1981
C

COMPUTER SCIENCE DIVISION

Electrical Engineering and Computer Science Department
U-157
The University of Connecticut
Storrs, Connecticut 06268

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

81 4 13 136

(2)

DECENTRALIZED SYSTEMS

E. Balkovich

Technical Report: CS-15-80
Contract: DASG60-79-C-0117

DTIC
ELECTE
MAY 20 1981
S
C.

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

The University of Connecticut

STORRS, CONNECTICUT 06268

THE SCHOOL OF ENGINEERING

March 31, 1981

The attached technical report is the Final Report on the Research Grant #DASG 60-79-C-0117, Research in Decentralized Systems, U. S. Army Contract. E. E. Balkovich, Principal Investigator

(9) Final Rept. 1 Sep 79-31 Dec 80

(10) E.E. Balkovich

(6)

DECENTRALIZED SYSTEMS

Technical Report: CSR-15-80

Contract: DASG60-79-C-0117

(15)

For the Period

1 September 1979 - 31 December 1980

(11)

(12) 237

Performed for

Ballistic Missile Defense Advanced Technology Center

P.O. Box 1500

Huntsville, AL 35807

(14) TR-CSR-15-80

Performed by

Electrical Engineering and Computer Science Department

University of Connecticut

Storrs, CT 06268

Authors: See Table of Contents

Principal Investigator: E. Balkovich

Accession For

NTIS GRA&I

DTIC TAB

Unannounced

Justification

By

Distribution/

Availability Codes

Dist Special

A

TR-CSR-1

DISCLAIMER

The views and conclusions contained in this report are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Ballistic Missile Defense Advanced Technology Center, Huntsville, Alabama, or the U.S. Government.

TABLE OF CONTENTS

Part I

Introduction and Overview

by

E. Balkovich

1.0 INTRODUCTION

In September 1979, the University of Connecticut under contract DASG60-79-C-0117, initiated a research program in decentralized systems. This work was motivated by the observation that software architectures, based on collections of cooperating, concurrent software processes or tasks, could be applied to logically distribute applications such as BMD.

This approach to distributing the software of an application offers a potential advantage in areas such as fault-tolerance. For example, it is possible to distribute information structures so that they remain generally accessable in spite of failures affecting component parts of the structure. Furthermore, such gracefully degrading software structures do not incur the same run-time costs as roll-back and recovery strategies. This approach is relevant to the BMD problem which requires fault-tolerant application software that is constrained to operate in a run-time environment with severe time constraints.

An operating system is required to provide run-time support for the software processes that would be used to distribute a BMD application. This method of decentralizing software functions appears to make extensive use of these operating system functions. Large numbers of processes are required. Typically, each process is relatively small (measured in terms of execution time and storage requirements) and interacts frequently with other processes to cooperatively implement the functions needed by an application.

1.1 RESEARCH OBJECTIVES

This research had two objectives: (1) to investigate implementations of the programming concepts needed to decentralize application software, and (2) to examine the performance of such implementations.

The run-time environment of decentralized application software requires the implementation of software processes and mechanisms that al-

low processes to interact. This run-time environment is generally implemented by a portion of an operating system known as the kernel. The principles governing the construction of a kernel are well understood for a conventional, single-processor system. One goal of this work was to develop principles for distributing the implementation of kernel functions among the processing elements of a distributed computer system. These principles were to be based on a partition of state information (used to control software processes and their interactions) and a communication protocol that would allow separate instances of the kernel (resident at different processing elements) to cooperate in implementing process interactions.

It was expected that distributed application software would use the facilities of an operating system kernel in a qualitatively different way than software written for a single processor system. Thus, the second goal of the investigation was to use a prototype implementation of a distributed operating system kernel to investigate how distributed application software would use a distributed computer system. This effort concentrated on characterizing the nature of software processes used to distribute an application and how these processes used the functions of the operating system kernel and the interconnection structure of the distributed computer system. The latter portion of this effort was limited to loosely coupled systems that could be constructed using the experimental facility available at the University of Connecticut.

1.2 RESEARCH FACILITY

The research facility used by this project reflects key views on the decentralization of application software. The facility included both hardware and software. The following summary of the hardware is intended to illustrate the types of loosely coupled systems considered by this investigation and to provide backgound for later sections of the final report. The summary of software used by this investigation is intended to review the general structure of distributed applications that was assumed by this study. It also summarizes some of the specific

tools used to support this study.

1.2.1 Hardware

This research was supported using an existing distributed computer system. This system was constructed from five Digital Equipment Corporation LSI-11 microcomputers hosted by a larger PDP-11 system. The host system provides support for software development and analysis of performance data obtained from the distributed computer system. Initially, the microcomputers were loosely coupled using a point-to-point communication network based on standard serial interfaces. This provided a low-cost, flexible interconnection structure that could be used to support investigations of alternative topologies. Later, a second communication subsystem, based on a contention bus, was added. The second subsystem provided more sophisticated hardware support for communication functions and a media that naturally supports broadcasting.

Each processing element of the distributed computer system uses a fully configured LSI-11 microcomputer. The basic processor-memory configuration of each element is identical. Each processor has a local memory of 28K-words. The console device of each computer has been configured as a port in a four-port, asynchronous serial interface. One computer differs from the other processing elements. It has access to a programmable real-time clock and a double-density floppy disk.

The experimental facilities are hosted by a PDP-11/60 system. Five serial interfaces of this system are connected to the system consoles of each processing element of the experimental facility. The transmitter of the console device of each element is routed to both the receiver of its corresponding PDP-11/60 interface and to a connector on the console panel of the experimental facility. Thus, the PDP-11/60 and a dedicated terminal can be used used to monitor transmissions from each processing element. The receiver of the console device of each element is routed through a switch that can be used to connect it to either the transmitter of its corresponding PDP-11/60 interface or to the keyboard

of a dedicated terminal. This switch can be used to select a local terminal or the host facility to provide the inputs for a processing element.

There are two mechanisms available for measuring the performance of software executed by the experimental facility. These are: the serial interfaces to the PDP-11/60 system and the programmable real-time clock available at one of the microcomputers. The programmable clock provides rudimentary clock facilities to all the processing elements of the system. It generates an externally accessable overflow signal when the clock is in operation. This signal is routed to the external event line of all processing elements including the processor controlling the clock. Thus, a periodic source of interrupts can be generated at each element. These interrupts can be used to sample local state information or to maintain a measure of elapsed time.

1.2.2 Software

This work was supported by a number of software development tools. The host computer system, and its general purpose operating system, provided an environment for developing and evaluating experimental software written for the distributed computer system. Due to an earlier collaboration with researchers at the University of Warwick, this research project had access to a compiler for an experimental programming language for distributed computing (EPL). This project adopted EPL and applied it in two ways: (1) to write benchmark application programs for the distributed computer system, and (2) to define a virtual machine that was implemented by the operating system kernel of the distributed computer.

The major features of EPL are:

(1) An EPL program executes a number of autonomous software processes called actors. The code executed by a process is defined

by a sharable unit called an act.

(2) Software processes (actors) interact with each other by sending messages. It is not possible to share variables. Messages serve to both communicate information and to synchronize processes (actors).

(3) Process definitions (acts) may be nested. Processes (actors) can be created to provide parallelism and completely general networks of processes.

(4) The basic data object of EPL is a word. The language is typeless so that a word may be regarded as a bit-pattern, a number, or the name of a process (actor). In this respect, EPL has both the flexibility and limitations of machine language.

These language features provide programming concepts that are representative of the mainstream of thinking about software structures for distributed computing. The simplicity of this language makes it well suited to experimentation since it can be easily modified to investigate new ideas.

One of the design goals of the experimental distributed computer was to exploit the facilities of the host computer operating system when developing software for the distributed computer system and when observing or interacting with experimental software. The ability to operate the entire distributed computer system from a single host terminal was a specific objective. This required host software that could monitor and display all data transmitted from the experimental facility, dynamically switch a single terminal between the console devices of the five processing sites, and provide suitably formatted inputs for the microcode ODT loader of LSI-11 systems.

Software was developed for the PDP-11/60 to satisify these specific requirements. When invoked, this software generates processes that monitor and record, in separate files, all data received from the process-

ing elements of the experimental facility. These data are also displayed at an user terminal. The output of a terminal keyboard can be switched to provide input for each of the processing elements of the experimental facility. These inputs can be used to execute any of the commands recognized by the microcode ODT of the LSI-11 systems or to supply data for software executed by the experimental facility. One of the microcode ODT commands invokes a loader that can be used with a serial interface. The PDP-11/60 software includes an absolute loader that can be used to read loadable files prepared at the host facility.

A major item of analysis software developed for the host computer system was a trace driven simulation of EPL programs. This simulation is driven by a dynamic trace of all process interactions occuring as a benchmark EPL program is executed. The simulation was designed to evaluate how a particular EPL program would be executed by distributed computer systems differing from those that could be configured in the laboratory. Many of the key parameters used in the performance models of this simulation (e.g., execution time requried support a particular type of process interaction) were based on the experimental operating system software developed for the laboratory facility.

2.0 SUMMARY OF RESULTS

Three major results were obtained from this work: (1) the design of a prototype distributed operating system kernel for a loosely coupled distributed computer system, (2) refinements to the programming concepts (supported by the operating system kernel) that can be used to express application software, and (3) preliminary evaluations of the performance of such software.

2.1 OPERATING SYSTEM KERNEL

Part II of the final report describes the design and implementation of a prototype operating system kernel for a distributed computer system. This operating system is specifically designed to provide run-time support for distributed software written in the programming language

EPL. The design of the operating system kernel is based on the principles that were outlined in section 1. The implementation of the prototype of the design was done using a loosely coupled computer system. The communication network supporting this system was a full point-to-point serial interconnection of the computers.

A key facet of the operating system design was a communication protocol. This protocol was the mechanism used by instances of the operating system, executed by different processing elements, to synchronize software processes and to transfer data between processes. The initial protocol was chosen to be as simple as possible. It is asymmetric and always initiated by the processing element representing the sending process.

The algorithms of the design are more complex than those required to implement a functionally equivalent operating system kernel for a uniprocessor. This increase in complexity is due to the lack of complete state information at any processing element, and the protocol required to overcome this limitation. The implementation of the design is significantly slower than the implementation of a functionally equivalent operating system kernel for a uniprocessor. Delays are introduced by the more complex algorithms and by the communication subsystem.

Another important feature of the design was its encapsulation of the communication subsystem hardware. Outer layers of software (the operating system kernel algorithms and application software written in EPL) remained unaware of how the communication subsystem was actually implemented. This encapsulation of the communication subsystem allowed the algorithms of the operating system kernel to uniformly implement process interactions and provided for a family of operating system kernels. This family of operating system kernels anticipated variations in the hardware that could be used to implement the communication subsystem and variations in the functions implemented by the operating system kernel.

The provision for a family of operating system kernels was exercised by implementing a second version of the operating system kernel supporting EPL. The second version implemented the run-time environment required by the extensions to EPL discussed in Part IV of the final report. This design used a more complex, symmetric protocol required to detect faults at the processing elements.

The concept of information hiding represents one of several software engineering principles that can applied to the design of the operating system kernel. The application of such principles, with particular emphasis on the use of data abstraction, is discussed in Part III of this report.

2.2 REFINED PROGRAMMING CONCEPTS

The programming language EPL was designed for experimentation with software for distributed computer systems. This research used it to define the virtual machine that was implemented by the operating system kernel and to write applications that would use the distributed computer system. Initial experience using this language and the operating system supporting it indicated the need for a number of refinements to the programming concepts embodied in the language.

It was virtually impossible to design a general purpose scheduling algorithm for the operating system kernel that would anticipate the individual needs of applications. It became clear that a general approach would be to place static scheduling decisions (determining the residence of processes when they were created) in the domain of the application software rather than the operating system kernel. This observation motivated a programming concept that would allow the processing element to be specified when a process was created.

In attempting to implement published algorithms for decentralized software, it also became clear that programming concepts were needed that would allow a process to detect and respond to failures associated with other processes in an application. This led to further modifica-

tion of the programming concepts dealing with process interactions.

Part IV of the final report describes the programming concepts proposed to deal with these issues. It does so by proposing refinements to the programming langauge EPL. These refinements are described in terms of their impact on the syntax of the language and their impact on the virtual machine required to provide run-time support for programs written in the language. The majority of these changes introduce programming concepts that allow an operating system kernel to report failures occurring during process interactions. Part IV of the final report discusses the events that should be reported to an application program as faults. It also discusses approaches to implementing these programming concepts in an operating system kernel.

These refinements were incorporated into the translator for EPL. As noted earlier, a second implementation of the operating system kernel was also generated to support programs written in this revised version of EPL. These refinements resulted in the use of a more complex, symmetric protocol to coordinate instances of the operating system kernel.

2.3 PERFORMANCE EVALUATION

The prototype operating system kernels, and benchmark programs (written in EPL) were used to explore performance issues. This exploration addressed the nature of software for decentralized systems and how that software would use specific hardware components. It tried to quantify the features of software written to decentralize systems. It also tried to assess how that software would use the hardware interconnection structure linking the processing elements of a distributed computer.

Initially, measurements of the characteristics of benchmark software written in EPL were obtained. One measurement of particular interest was the use of operating system functions by EPL programs. These measurements were then used to interpret existing performance models of specific hardware interconnection structures. This provided insight into how particular interconnection structures could be expected

to behave when used to implement decentralized systems. Finally, trace-driven simulation techniques were used to compare the behavior of benchmark EPL programs executed by different distributed computer architectures. The results of these investigations are reported in Parts V-VII of the final report.

The measurements indicate that the software structures used to decentralize systems will extensively use the concept of a software process to distribute the functions provided by software. The grain of computation represented by a process can be expected to be considerably smaller than that used in a conventional multiprogrammed operating system. In addition, multiple processes will generally cooperate to provide a specific service, or access to an information structure. As a result, the software can be expected to use the functions provided by the operating system kernel very frequently. These observations are discussed in Part V of the report.

Architectures based on the use of a shared contention bus are likely to be a preferred design for loosely coupled systems because of their simplicity and regularity. For this reason, a major effort was made to examine how software for a decentralized system would use such an interconnection structure. These results are discussed in Part VI of the report. The characteristics of software used to decentralized systems were used to interpret a number of existing models of contention bus performance. The small size of processes and the use of a communication protocol by the operating system lead to a distribution of message lengths that favors short messages. This particular distribution of message lengths utilizes a small fraction of the bandwidth of a contention bus. This limitation suggests that systems be designed with adequate bus bandwidth or that alternative algorithms for using the bus be developed to improve bus utilization.

The hardware facility of this research program supports experimentation with only loosely coupled systems. Trace-driven simulation techniques were used to explore how software for decentralized systems would

perform with system configurations other than those that could be constructed in the laboratory. The simulation was driven by a trace of process interactions obtained by actually executing a benchmark EPL program. This trace of events was combined with timing information to determine how the same set of events would occur in different system architectures. These results are preliminary, but they do suggest a number of architectural issues that should be explored and a methodology for doing so. This work is presented in section VII of the report.

Part II

A Distributed Operating System Kernel

by

S. Fontaine

CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The primary focus of this investigation is the structure and organization of software to control distributed systems based on loosely coupled, multiple processors. It is assumed that there is no shared memory and that all processors are directly connected by one or more communication links.

There is a consensus among researchers that the concept of a process can be applied to decentralize and distribute software. Collections of processes cooperate to accomplish a task. Primitives for interprocess synchronization and communication are essential to support process cooperation.

A particular model of processes was chosen for further study. In this model, all processes are concurrent and independent once created. Process communication and synchronization are combined in a message passing construct. Messages are not automatically buffered.

The design and performance characteristics of the operating system kernel, are of special interest. The kernel provides the virtual machine to support the process model and hardware transparency to conceal the machine configuration from the high level processes. The performance of such a system is a tradeoff between the amount of parallelism possible versus the additional costs incurred by the distribution of processes.

There has been much discussion about exactly what constitutes a distributed system. The definition used here, proposed by Enslow[ENSL78], is presented as a set of requirements which a distributed system must satisfy. It should possess:

1. A multiplicity of general purpose resource components, including both physical and logical

resources, that can be assigned to specific tasks on a dynamic basis.

2. A physical distribution of these system components interacting through a communication network.

3. A high level operating system that unifies and integrates the control of the distributed components. Individual processors each have their own local operating system, and these may be unique.

4. System transparency, permitting services to be requested by name only. The user should be able to request an action by specifying what is to be done and not be required to specify which physical or logical component is to provide the service.

5. Cooperative autonomy, in the operation and interaction of both physical and logical resources.

Several of these requirements are met by features of the language EPL [TAYL][MAY79] and by the system hardware. Others are met by the kernel. EPL is based on a process model which supports dynamic software configuration and multiple process instances. These processes cooperate to accomplish a single task. The EPL constructs for process cooperation hide the location of processes so that they could be on a single node or distributed across many. The hardware is a network of interconnected microprocessor/memory pairs. These points are sufficient to fulfill the requirements for multiplicity, distribution and cooperative autonomy at both the logical and physical levels.

High level system coordination could be implemented by either EPL programs, the kernel, or a combination of the two. Presently there is no high level operating system, but the kernel does support system wide policies for the essential functions such as memory management and scheduling. Later, some of these and other resource management facilities will be included in an operating system written in EPL.

The fifth requirement of the definition, that services be requested by name only, is not

completely met by the system. EPL has no built in capability for referencing processes by service provided. However, this capability could be added by including a special name manager process which keeps lists of process names indexed by service (similar to the switchboard in DEMOS [BASK77]). A user needing a particular service could get its name from the name manager and then communicate directly with the service process. The kernel does provide hardware transparency, which enables EPL processes to be ignorant of the actual physical component providing a service.

CHAPTER II

BACKGROUND

Before attempting to study the problems involved in building distributed systems, the concepts underlying the problems and possible solutions were examined. Systems and languages currently under development for distributed processing were studied and compared with one another to find a consensus on the central issues and to view a spectrum of proposed solutions. A language which addressed these issues and whose solutions are representative of present research, was chosen as the implementation language for the project.

Each of the systems and languages examined defines a process model as the basic unit of software organization. The processes of a program cooperate to accomplish some task. Mechanisms must be provided to allow the processes to synchronize and communicate with one another. Because the processes are distributed and control is decentralized, the interactions between processes are nondeterministic. The languages studied reflect this by allowing nondeterminism in certain language features. The basic concepts in distributed processing emphasised here are the use of a process model as the basic software unit, communication and synchronization between processes, and nondeterminism allowed for process interactions.

2.0.1 Process Model

The use of many small processes as a building block to implement a program provides a convient unit for the distribution of software. Since these processes may be on separate processors they are designed to be autonomous entities. Interactions between processes are well defined and independent of the actual distribution. Between interactions, processes on separate processors are capable of running in parallel. In summary, the process requires no centralized control and is the software unit of distribution and parallelism.

2.0.2 Communications

The ability to exchange information is essential for process cooperation. Because the run time configuration of the software is variable, the actual configuration should be transparent to process communication. Two methods in particular have been widely accepted: shared memory and message passing. Shared memory requires specialized hardware, but message passing can be implemented on shared memory machines as well as on machines with disjoint memory. Message passing is therefore considered to be the more flexible of the two. The main role of process communications is to provide configuration transparency for the passing of information between processes.

2.0.3 Synchronization

Because processes are autonomous, and have no centralized control, some means must be provided for process synchronization. For example, a process which is a device handler must have control over when it accepts requests, prehaps how many it accepts, and from whom. Also, a process using the service might need to wait for its completion before continuing. A mechanism for synchronization is necessary to maintain a coherent timing structure between the processes.

2.0.4 Nondeterminism

Since processes exist on separate processors, each with its own environment, the absolute timing of a particular process cannot be predicted. If the timing of a single process cannot be predicted, the neither can the timing of interactions between processes. For example, a process which is to be used by several other processes cannot always guarantee beforehand the ordering of requests. Because of this, decentralized and distributed software is inherently nondeterministic and a distributed system language must be designed to tolerate some degree of nondeterminism in the interactions of processes.

Each of the systems and languages described in the following paragraphs defines a process model with communication and synchronization capabilities. These three seem to be the

central programming language concepts needed to distribute computations. In the following discussion, particular emphasis is placed on differences in the solutions proposed to implement these concepts.

2.1 LANGUAGES

2.1.1 Communicating Sequential Processes

The main objective of Communicating Sequential Processes (CSP) [HOAR78] is to define a single, simple solution to both process communication and synchronization. The text of a CSP program places an upper limit, at compile time, on the number of processes which can exist at run time. Multiple instances of a process are treated as an array of processes: each process being distinguished by its integer subscript. As in a standard programming language, the maximum number of array elements is specified. Since the maximum number of multiple instances is known and the single instances can be counted, the upper limit of processes for the CSP program is known at compile time. Processes are created dynamically and terminate at the completion of their code. There is no special kill or self destruct mechanism in CSP. A new process, or child, is entered into the system at the request of an already existing process, called the parent. With the parallel command, a parent can simultaneously create several children which run concurrently. The parent process cannot continue execution until all of its children have terminated, and children are in turn blocked until their children complete. CSP processes are structured into a strict hierarchy based on parent-child relationships.

Processes communicate by sending messages. Only two processes can be involved in a single communication and each must specify the other's name. Names are assigned at compile time. The process which is the source of the message executes an output command, and the destination process has a corresponding input command. Messages are not automatically buffered, which means that both processes must be prepared to handle the communication at the time it occurs. To guarantee this, the process which becomes ready first is blocked from running until the other is ready too. The message is sent and then both

continue. If buffering is needed, then it can be implemented by the user with a buffer process.

Synchronization between a parent and its children is implicit in the fact that the parent is delayed until its children complete execution. The synchronization of processes at a single level in the hierarchy is implicit in the communication construct. To avoid buffering of messages, the two communicating processes are required to be at a particular point in their code. Therefore, the two processes are synchronized at the time the communication occurs. This second mechanism can be used as the basis for constructing other means of synchronization, such as semaphores.

Nondeterminism is introduced and controlled in CSP through the use of guarded commands. A guarded command has two parts: a guard, which is executed first, and a command list. The command list is executed if and only if the execution of its associated guard does not fail. An alternative command consists of a series of guarded commands from which one of the command lists with a true guard is chosen for execution. If all of the guards fail, the alternative command fails. A variation of this is the repetitive command which repeats an alternative command until all guards are false. CSP allows input commands to appear in a guard. An input command succeeds if the corresponding output command is executed, and fails if the source named in the input command has terminated. A repetitive command which includes as part of its guard a series of input commands is effectively receiving a message from a set of possible senders. This would probably be implemented with a first come first serve algorithm. If none of the named sources are prepared to send, but are still executing, then the receiver process is delayed until one of the guards succeeds.

2.1.2 Distributed Processes

Distributed Processes (DP) [HANS78] is designed for real time applications on a microprocessor network with distributed storage. The DP process model has been affected by the special requirements of real time applications. To handle the high speeds and demands associated with real time, only one process is to exist on each processor. At compile time, the

fixed number of concurrent processes are created and at run time they are started simultaneously. As in CSP, the processes can be structured in an array.

A process has two basic operations: an initial statement and processing of external requests. These operations are interleaved starting with the initialization. When the initial statement is either completed or waiting, the process is prepared to handle requests form other processes. If the operation for a particular request has in turn been terminated or put into a wait, then either a preempted operation is continued (possibly initialization) or a new request is accepted. If the initialization operation terminates then the process simply accepts requests.

The operations being requested are in the form of procedures. Procedures can contain input/output parameters, local variables and statements. A process making a procedure call is delayed until the operation is completed. This arrangement is similar to the interactions found in a hierarchy of processes, however, since all DP processes are created and started running simultaneously and never terminate, it cannot be described as a hierarchy.

The only communication between processes is in the form of procedure calls. Information is passed by value to the procedure in the input parameters and returned by value in the output parameters. The passing of values could be implemented by either message passing or shared memory.

DP introduces and controls nondeterminism with two types of guarded statements: the guarded command and the guarded region. These constructs are similar in intent to the CSP guarded command described earlier, but function somewhat differently. CSP allows input commands in guards. If the process named in the input command is still active, but no message has arrived the guard neither succeeds nor fails and the process is delayed until the decision can be made. However, a DP guard is a conditional expression based only on the state of the process's variables, which means that success or failure is immediately evident.

A DP guarded command consists of a series of guards each associated with a command list (similar to the CSP alternative command). One of the command lists with a true guard is

executed. If no guards are true, the guarded command fails. A guarded region is structurally the same as a guarded command, but if none of the guards is true, instead of the command failing, the process is blocked until at least one guard has become true.

There are two methods for synchronization provided by DP. The first is implicit in the procedure call. When a process makes a procedure call, it is not allowed to continue execution until the call is completed. This assures synchronization between the two processes. The more general method is the guarded region described in the previous paragraphs. The guarded region can be used to implement control structures, such as a semaphore, to synchronize processes. The semaphore could in turn be used to implement more complex control structures.

2.1.3 EPL

EPL (Experimental Programming Language) [TAYL][MAY79] is designed as a systems language for distributed systems. It encourages a style of programming which uses many small cooperating processes. The code for a process is declared as an act and an instance of that act is called an actor. An act is designed to be re-entrant so that multiple instances of an act can share the same code. Actors are created dynamically and terminate when the end of the code is reached. EPL allows actors to be created simultaneously. Unlike CSP and DP, there is no compile time upper bound placed on the number of actors that can exist in the system.

An actor wishing to create a new actor, specifies the act and makes a CREATE call to the kernel. The actor making the call, referred to as the parent, is given the new actor's, or child's, name and is allowed to pass parameters to the child before the child begins execution. Beyond this point, there is no special connection between parent and child actors. EPL does not enforce any strict relationship, such as a hierarchy, between actors.

Both process communication and synchronization are handled by the message passing (SEND/RECEIVE) construct. Only two processes, a sender and a receiver, can be involved in

a communication. The sending actor must name the receiver, but EPL allows the receiver to either specify the sender by name, or to receive from any sender. Therefore, the message passing can by completely deterministic, with both of the actors involved naming the other, or nondeterministic, with the sender specifying the receiver but the receiver being capable of accepting from any requesting sender. EPL does not include a special mechanism for receiving from one of a set of specified senders. However, if necessary, the user can structure the process interactions to simulate this capability.

Synchronization is handled by causing either the sender or receiver, whichever is prepared for communication first, to wait until the other is ready. With this mechanism, it is also unnecessary to buffer the message, since it can be moved directly from the sender's space to the receiver's space. EPL provides three ways by which a process may discover another's name: 1) a parent knows the name of its child process, 2) a child can receive a name as a parameter from its parent during initialization, or 3) the name could be sent as part of a message. A process does not automatically know its own name, but must discover it in one of the methods mentioned.

2.2 SYSTEMS

Several systems currently under development were studied to better understand a kernel's characteristics and role within a distributed system. In each of the systems, the kernel acts as the interface between the hardware and the supported language constructs. This enables the high level language to remain ignorant of the underlying hardware. Because it is so dependent on the hardware and the language, the kernel for a particular system can be characterized by describing these two aspects.

The software in the systems examined are each in two sections: a kernel to provide low level support for the distributed language primitives (process operations such as creation and

communication) and a high level set of utilities (resource managers, debugging aids) written in the distributed language. Several of these systems have been selected for further description to represent a wide range of view points. Since the major concern of this project is software and not hardware or system utilities, more emphasis is placed on the programming language concepts. As in the previous discussion of languages, the most important language constructs are the process model definition,and interprocess communication, synchronization and nondeterminism.

2.2.1 STAROS

STAROS [JONE79] is a software kernel and set of utilities for a system called Cm* with approximately fifty tightly coupled processors and shared memory. The processors in Cm* support capability checking, which is one of the major goals of STAROS. Another of the main objectives of STAROS is to explore the feasibility of and potential benefits from their process model which is termed a task force.

The processes in a task force are created dynamically and either terminate at the completion of the code, cycle endlessly or are terminated by a kill command in another process. Multiple instances of a process can share data objects. STAROS supports two different types of process relationships: dependent - in which the processes form dependency trees based on a parent-child hierarchy, and independent. If the root process of a dependency tree is killed, the tree dies as well.

In STAROS, process communication is based on message passing. Since the nodes in Cm* share memory, the message passing amounts to copying from one memory location to another. Messages are automatically buffered in an object called a mailbox so the sender is not required to wait for the receiver to become ready to receive. The receiving process need not wait for a sender either. If the message is not found in the mailbox , the receiver has explicit control of whether or not it waits.

Synchronization is handled explicitly by a process. STAROS has an event primitive which a process can use to wait for some condition to become true such as receipt of a message.

2.2.2 ROSCOE

ROSCOE [SOLO79] is a general purpose distributed computation resource implemented on a loosely coupled network of five LSI 11s with no shared memory. Both a kernel and high level utility processes are included. Communication is based on links which represent a one way connection between two processes. The owner of the link is the receiver, and the sender is called the holder of the link. A holder can duplicate the link or give it away to other processes by passing its name as part of a message. An owner can request to receive from any one of a set of its owned links. Buffers are used by the I/O system to hold outgoing as well as incoming messages which implies that the link holder is not required to wait for the owner to be ready to receive.

2.2.3 HXDP

Like DP, HXDP [BOEB] is designed for support of real time applications. Built at the Honeywell Systems and Research Center, it is based on loosely coupled memory processor pairs and special message passing hardware. Communication between processes is implemented with buffered message passing. All knowledge of the hardware configuration is within the kernel so network shape and process location are transparent at the process level. Synchronization is explicit and based on an event primitive.

2.3 SUMMARY

Current research in distributed processing has focused on several programming language concepts as solutions to problems in distributing software. However, there is disagreement on the specific implementation of these features.

Every language examined uses a process model as the basic software unit. Functionally, a process is on the same level as a subroutine in a standard language. The process provides a

convient unit for software distribution, the decentralization of control and parallelism. Processes behave as autonomous entities cooperating to accomplish some task.

In several of the systems processes are created dynamically, in others they are created at compile time. The processes can last forever or terminate either as a result of reaching the end of the code or by being the subject of a kill command. The maximum number of processes to exist can be defined at compile time or be limited only by the available resources. The relationship between processes can be an enforced hierarchy or dependent only on the programmer. Processes interact in well defined ways.

For processes to cooperate effectively, some means of exchanging information must be provided. The most important aspect of a communications method is that it be capable of concealing the hardware configuration. All of the proposed communication mechanisms described can be implemented with a message passing scheme. Message passing is a powerful and general tool for constructing interprocess communications.

Some of the communication constructs provide automatic buffering of messages and others rely on the user to provide buffering if it is necessary for the application. If the scheme uses unbuffered messages, then both processes must be prepared to communicate at the time the message is passed which requires that the process ready first be blocked. With automatic buffering, only the receiver needs to wait and the sender is not affected by the state of the receiver.

Researchers agree that some kind of synchronization between processes is necessary to structure autonomous processes into a coherent program. Since control is decentralized, a mechanism is required for process coordination.

Two distinct solutions for process synchronization can be recognized. One is based on an event primitive, in which the process waits for some condition to become true, and the other is an extension of the message passing concept, in which whatever process becomes prepared to communicate first is blocked until the other is ready. The method associated with message

passing can be used to implement the event primitive and provides a general and simple solution for the exchange of timing information.

Each of the languages has a control structure intended to handle the nondeterminism inherent in distributed systems. When several processes are capable of sending messages to the same process, that process cannot necessarily predict which of the senders will be prepared to communicate first.

Each language has a slightly different approach for dealing with this special case. Several of the languages use guarded commands to block a process until at least one of a set of conditions becomes true. These conditions could include input commands or event primitives keyed to a message arrival.

In the systems examined, the kernel acts as the interface between the high level program language and the hardware. Its major function is to support the primitives of the high level language in such a way as to provide hardware transparency. Because of its role as interface, the kernel is extremely sensitive to both the hardware and language characteristics. The hardware for the systems described range from those with off the shelf hardware to special purpose components, and shared memory to disjoint memory. The programming language concepts used by these systems have been included in the language summary.

EPL was chosen for use in this project for several reasons. It is representative of the state of the art in distributed systems languages. Its designers have attempted to keep the language simple and elegant by using the most general solutions to the major issues. There is no hierarchy of processes, both timing and data exchange are managed by the message passing construct, and a nondeterministic control structure is included for process interactions. EPL's structural simplicity and minimum number of primitives is reflected in the size and complexity of the kernel which means less expense is involved in exploring alternative kernel designs. An EPL compiler designed to be easily modified and compatible with the research facility was available.

CHAPTER III

ENVIRONMENT

The major purpose of an operating system kernel is to provide a virtual machine for the high level programming language. The kernel acts as the interface between the high level language being supported and the system hardware. The characteristics of the hardware and high level language, considered here as part of the kernel environment, have great influence on the design and implementation of the kernel. Another important aspect of the environment is the development tools. These tools, such as compilers and hardware facilities, have had some impact on kernel structure as well as on performance measurements.

First, an overview of EPL is given, similar to that in Chapter II. The features of EPL actually supported by the kernel, called primitives, are described in some detail. Following this, are the EPL run time requirements, such as memory specifications and the format for EPL to kernel communication. The research facilities, including the hardware and tools, are then described.

3.1 EPL

EPL is based on a process model in which processes are created dynamically and coordinated with two party, synchronous message passing. An EPL process is called an actor and the code for an actor is called an act. When a new actor is created, the parent actor is provided with the name of the child so that it can send initialization arguments. An actor terminates at the completion of its act.

The message passing construct is used for passing both data and timing information. The source actor is called the sender and the destination actor is called the receiver. Messages are not automatically buffered within the kernel, so the member of the pair prepared to communicate first must be blocked until the other is ready as well. The sender must know the

name of the receiver, but the receiver can either name a specific sender or receive from any sender. Actors are assigned names at run time. There are three ways for an EPL actor to discover the name of another actor: 1) a parent knows its child's name, 2) names can be passed to a child as initialization arguments, and 3) names can be included in a message.

The basic operations needed to support the EPL process model have been grouped into a set of primitives implemented within the kernel. When an actor needs to execute a primitive the kernel is called. Occasionally, primitives are referred to as kernel calls. A summary of the EPL primitives follows.

1. INIT - This primitive is not actually called by the EPL code, but by the startup routine. INIT takes care of whatever system initialization is necessary.
2. CREATE - CREATE is used by a parent actor to enter a new actor, or child, into the system. The child is allocated space on a processor but is not started until the parent executes the RUN primitive.
3. RUN - The RUN primitive allows a parent actor to pass initialization arguments to the newly created child. The child is then added to the local set of active processes.
4. TERMINATE - An actor executes the TERMINATE primitive when it completes the code for its act.
5. SEND - A SEND call is made by an actor to transmit a message to another process. If the named receiver is ready to communicate, the message is sent. Otherwise the sender waits.
6. RECEIVE - An actor needing information to continue execution uses the RECEIVE primitive. It can either specify a particular process as the message source, or receive from any process. If no appropriate sender is ready, the receiver waits.
7. SYSTEM - SYSTEM is for miscellaneous local operations which are not included among the major primitives. Currently, SYSTEM calls are provided to handle reading from and writing to terminals, and voluntary rescheduling of actors.

Memory is allocated to an actor at the time of its creation. The amount of memory required by an actor is static and the memory assigned to an actor cannot be accessed by any other actor. Every actor is allocated memory for two data structures: a process descriptor and a data segment. The process descriptor is of constant size and is used by the kernel to define the state of an actor and its relationship with other actors. Since the process descriptor is used by the kernel, the details of its implementation are discussed in Chapter IV and appendix [2].

Acts are designed to be reentrant so that multiple instances of an act can share the same code. Each actor is given a data segment to hold local variables called its runtime stack. The stack size required by each act is defined at compile time and is associated with the code for the act.

Before calling on the kernel to execute a primitive, EPL places any information needed into predetermined locations. Occasionally, EPL expects information to be returned from the kernel following the execution of a primitive. The interface between an actor and the kernel is dependent on the function being requested. Generally, the information is passed in the general purpose registers.

When an actor is executing, a pointer to the top of its stack resides in general purpose register R0 and serves as a base register. Certain primitives require the kernel to reference a location or set of contiguous locations within an actor's stack. For example, a SEND requires the message to be placed on the actor's stack. To send it the kernel must be told its starting location. When a kernel call requiring information on the stack is made, an offset to the start of the information is placed in another general purpose register (R1). If the stack reference can be more than one word, the kernel is given its length. Information returned from the kernel to the calling actor is placed in a general purpose register or directly onto the actor's stack. The following paragraphs describe the information layout for each primitive (for diagrams see appendix [1]).

1. CREATE - The parent actor names the act which the child is to execute. The act's name, defined as the start address of its code, is passed to the kernel in R1. The two words before the start of the act contain the stack size (in bytes) required for an instance of the act and the number of arguments (each one word) it expects to receive. After completion of the call, EPL expects the name of the child to be in R1.

2. RUN - An offset from the top of the stack to the start of the space to be referenced is in

R1. The first word in the stack space is the child's name. Following this, are the initialization arguments. The number of arguments to be sent is in the word before the start of the act and can be zero. After completion of the call, EPL expects the initialization arguments to be at the top of the child's stack.

3. SEND - An offset from the top of the stack to the start of the space to be referenced is in R1. The first word of the space is the name of the receiver. Following this is the message to be sent. The length of the message (in words) plus one word for the sender's name is known by the receiver.

4. RECEIVE - An offset from the top of the stack to the start of the space to be referenced is in R1. The first word in the space contains the senders name. If the sender's name is zero, it is interpreted as a RECEIVE from anyone. The following locations are to be used as a buffer to hold the message sent. The length of the message expected (in words) plus one word for the sender's name is placed in R2. Following a RECEIVE from anyone, EPL expects the name of the sender chosen by the kernel to be stored in the first word of the space referenced.

5. SYSTEM - An offset from the top of the stack to the start of the space to be referenced is in R1. The first word is the code for the function to be executed. The second is an optional argument. In the current version, the argument is used to hold a character for printing or a character read from the terminal.

Except when calling the kernel or the software multiplication/division routines (replaced by hardware), EPL does not use the system stack.

When creating a new actor, the parent names the ACT to be executed. Because creates can occur across machines, the ACT name must be unique and known system wide. In the present EPL, an ACT is referred to by its start address. Since this naming scheme is to be used across machines, all machines must have the save ACT loaded at the same location (see appendix[3] for the memory layout).

3.2 HARDWARE

The processors are DEC LSI 11s equipped with the standard LSI 11 instruction set. There is no real time clock. Each processor has 28k bytes of memory, none of which is shared. [BALK80]

The processors are directly connected by means of DEC DLV11J serial links. These links are asynchronous and bidirectional. They are capable of detecting certain errors on data received: framing, parity, and overwrite. Some alternative configurations, such as a ring and a doubly linked ring, are under consideration.

A DEC 11/60 running UNIX™ was used for preliminary testing of the kernel, down line loading to the LSI 11s, and general purpose tools such as the C compiler, editors etc. It was also used to collect and evaluate data on system performance.

CHAPTER IV

SYSTEM SOFTWARE

The system software is organized into three levels. The division and classification of the software is based on its function and the amount of hardware knowledge required by the function. The lower the level, the more hardware dependent are the functions performed. Information flows are defined both between adjacent levels and between units within a single level. Interlevel communication is described as being vertical and intralevel communication is called horizontal.

At the top level are the EPL actors. They see a single virtual machine provided by the second level, or kernel. A copy of the kernel resides on every processor. Each kernel is distinguished by a unique ID number. The kernel is ignorant of the link hardware details and it makes no distinction between requests from local and nonlocal actors. Knowledge of the link hardware and actor distribution has been isolated in the third level, called the I/O subsystem. The I/O subsystem is responsible for routing kernel requests to the correct processor.

The communication paths between these levels and within them is shown in figure [1]. Each of the vertical arrows and the horizontal arrow between I/O subsystems represent an actual information path. The horizontal arrows between kernels and EPL actors represent a virtual communication link. Each of the virtual paths actually follows the vertical paths to the I/O subsystem, across machines on the I/O subsystem horizontal paths, and back up again through the different levels. The role played by each of the levels is described in the following paragraphs. Next, the design and implementation details and communication protocols of the kernel and I/O subsystem are discussed. The memory layout and description of source code files are in appendix [3].

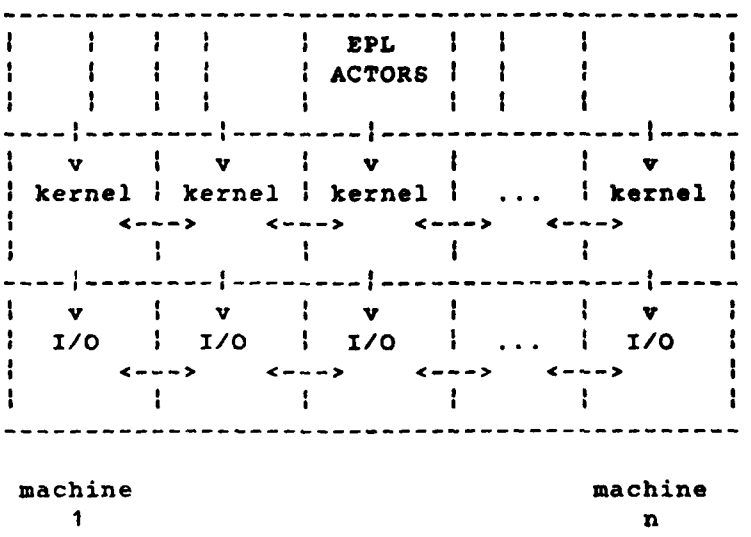


Figure 1. SYSTEM DIAGRAM

4.1 EPL

A high level description of EPL is in Chapter II and a more detailed one, including the EPL to kernel interface, is in Chapter III. EPL actors communicate vertically with the kernel and horizontally among themselves. The operations required to support the process model and process interactions have been grouped into a set of primitives which are implemented within the kernel. These primitives include such functions as actor creation, termination, and communication. Before making a kernel call, the EPL actor places information needed by the kernel in predefined places (the general purpose registers and the actor's data segment) and then executes a trap into the kernel.

The horizontal communications are handled by a synchronous message passing construct called SEND/RECEIVE. The mechanics for this construct are actually shared between the kernel and I/O subsystem, but logically it can be considered a direct connection between actors.

4.2 KERNEL

The kernel is responsible for providing the virtual machine required by the EPL actors. To do this, the kernel must fill three roles: 1) the interface between EPL and the hardware, 2) low level resource manager, and 3) actor manager. All of the hardware dependent functions

needed to support the top level EPL program are implemented within the kernel. These functions, included among the primitives, are described in Chapter III. The low level resource management implements both local and global policies for memory management and actor scheduling. Actor management primarily involves maintaining all actors in a valid state, and is not entirely separate from the two other roles.

The overall design philosophy was to keep the kernel as simple as possible. It was decided to optimize the kernel for logical simplicity, occasionally at the expense of efficiency. For example, the kernel could handle references to local actors more efficiently as a special case and pass nonlocal actor references to the I/O subsystem for routing to other processors. However, it is logically simpler to assign all routing of actor references to the I/O subsystem, including local references. Because of this decision, the kernel is able to handle all actor references in the same way and is ignorant of the actual location of a particular actor. Another concern was to keep dynamic memory requirements simple. This way, the problems associated with memory allocation, and complex schemes to solve them, could be avoided. Because of this, memory is not necessarily used efficiently.

4.2.1 System Information

Some of the most crucial issues in the design of a kernel revolve around the maintenance of system information. The biggest question is whether or not to replicate nonlocal information on each processor. It is possible that a system could use such information to simplify fault recovery as well as to reduce processor to processor enquiries. However, replicating information within the system introduces the problems associated with the maintainence of consistency in a replicated data base. Access to the data base must be protected to guarantee its integrity. It has been shown [FONT80] that replication of system information for an EPL kernel is not practical. Information about actors (contained in process descriptors) has been partitioned so that each kernel knows only about resident actors and must pass enquiries to other kernels for information about nonlocal actors. The kernel design relies on local information, enquiries,

and estimation instead of replicated information.

4.2.1.1 Data Structures

As explained in Chapter III, each actor is dynamically allocated memory for a run time stack (or data segment) and a process descriptor. The stack is controlled by the actor, but the process descriptor is maintained by the kernel. The process descriptor is used by the kernel to maintain an actor's state and its relationships with other actors, to pass information between levels and to save context when an actor is preempted.

Each EPL actor is assigned a unique name at runtime. EPL actors refer to one another by name and, on a kernel call, any actors pertinent to the execution of the primitive are specified by name. The kernel must take the name and use it to access any information about the actor it needs to complete the call. To simplify access to the information, it is placed in the calling actor's process descriptor. The name of an actor is defined as the start address of its process descriptor. The ID of the local kernel is placed in the low three bits of the name. Process descriptors are allocated at addresses where the low three bits are zero so that no addressing information is lost when the ID is added.

A process descriptor is sixteen words long. The first seven locations are used to save the values of the five general purpose registers, the program counter and the processor status register on a kernel call or interrupt. Several slots are provided for starting linked lists of process descriptors, and one slot for linking the process descriptor into a list. The status word holds the current state of the actor. Several other words are used to pass information to and from the I/O subsystem. The process descriptor is described in more detail in appendix [2].

The kernel maintains several queues of actors. The local set of all actors ready to run is contained in a structure called the ready list which is implemented as a circular linked list of process descriptors (figure [2]). This list is used by the kernel's low level scheduling and dispatching routines. The variable ready points to the last process descriptor in the queue. The top of the queue is pointed to by the chain link of the last process descriptor. The process

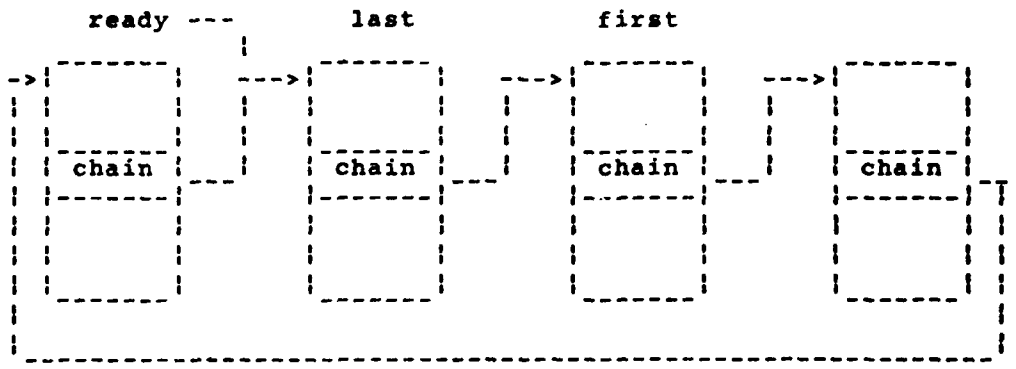


Figure 2. READY LIST

descriptor of the actor currently running is kept separately.

When an actor executes a RECEIVE, it is the kernel's responsibility to discover if the sender is prepared to send. If the RECEIVE is a RECEIVE from anyone, the kernel has no sender name to use in checking the sender's state. For this reason, the kernel maintains a queue for each actor, of actors ready to send it a message (figure [3]). If an actor wishes to SEND a message but the receiver is not yet prepared, the kernel links the sender's process descriptor to the receiver's. If another actor wishes to SEND to that same receiver, its process descriptor is added at the tail of the queue

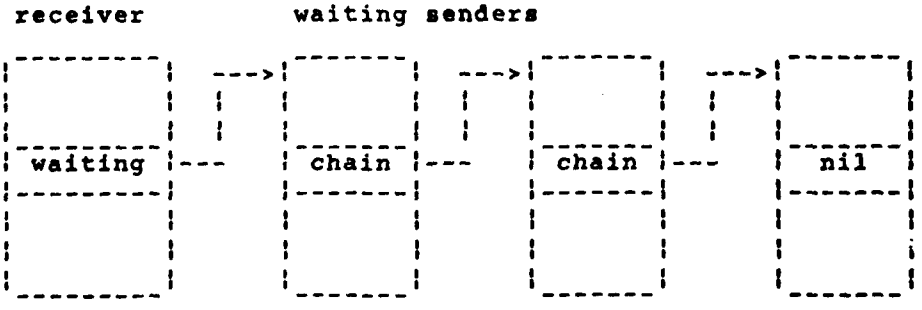


Figure 3. LIST OF SENDERS WAITING FOR A RECEIVER

When an actor executes a RECEIVE, its connected queue is checked for either the named actor, or any actor.

The distribution of actors across processors complicates this scheme. The sender may be on another processor and not have a local process descriptor which the kernel can add to the

linked list. One of the design decisions was to conceal the actual location of actors from the kernel so that local and nonlocal actors could be treated the same way. In such cases, a dummy process descriptor with the same size and structure as a regular one, is allocated locally to represent the nonlocal actor. Any information about the actor needed to process the request is placed in the dummy process descriptor.

Another list is needed to save the number of arguments expected by children created but not yet started running. EPL allows a parent to create several actors, save their names on the stack, and then start them running. This is so that the child actors can be passed one another's names as part of initialization. At the time a child is started running, the kernel must have access to the number of arguments expected by the child. Because more than one count may need to be stored at a time, a list is required. The information is placed in the child process descriptors which are then connected into a linked list attached to the parent's process descriptor (figure [4]). Any nonlocal children are allocated a dummy process descriptor.

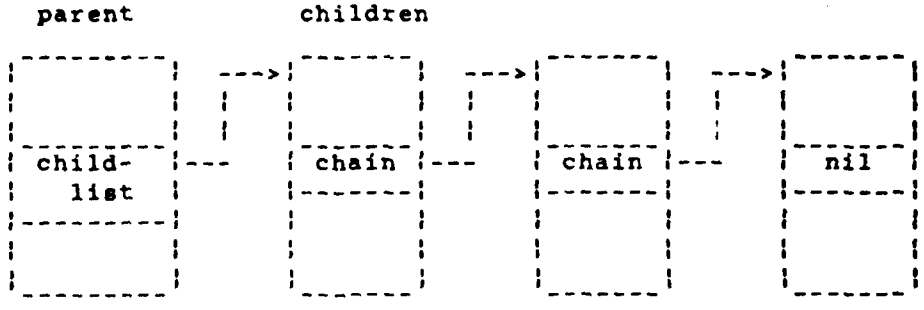


Figure 4. LIST OF UNSTARTED CHILDREN

4.2.1.2 Actor States

The behavior of an actor can be described by a finite state machine. The kernel has exclusive access to the actor states and its functions define all of the operations possible on a state. The kernel can be thought of as the actor state data abstraction. There are five basic states which a process can be in:

1. NEW - the process has been created but has not yet been enabled to run.
2. RDY - the process is in the local set of possibly running processes (i.e., it is on the ready list).
3. END - the process has completed execution and been terminated.
4. SND - the process has been blocked because the actor to which it wished to send is not yet prepared to accept the message.
5. RCV - the process has been blocked because no appropriate sender is ready to communicate with it.

The transitions between these states are shown in figure [5].

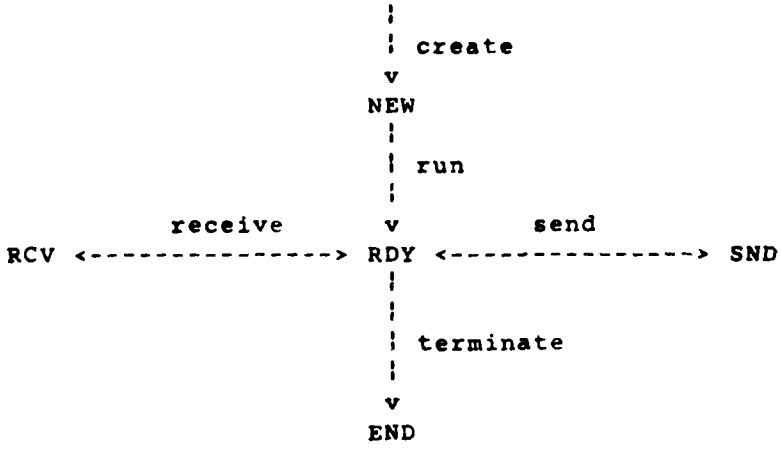


Figure 5. STATE TRANSITION DIAGRAM:

KERNEL CALLS ARE IN SMALL LETTERS, STATES IN CAPITOLS

The basic state transition diagram (used in a single processor version of the kernel) is complicated by the distribution of the actors. Additional states have been included to handle certain intermediate conditions, such as when the actor is blocked while its data segment is buffering information to be sent (parameters for a new process, or a message being sent as part of an EPL communication) or while the actor is waiting for an acknowledgement (a receiver has responded to a sender but the message has not yet arrived). The process state definitions for the distributed kernel are:

1. BLK - the process is waiting to receive a message and a suitable sender process has not yet enquired if the process is prepared to receive

2. CRT - the process is creating a child process and has not yet received the name of the child process
3. END - the process has terminated (the process may be referred to by other processes)
4. ENQ - the process has sent a message to enquire if the receiver process is prepared to accept a message
5. INT - the process is transmitting parameters to a child process
6. NEW - the process has been created (i.e., it has a name), but parameters have not yet been transmitted to it by its parent process
7. RCV - the process has acknowledged a sender process and is waiting to receive the text of the message
8. RDY - the process is logically ready to execute, but has not been assigned the CPU
9. RUN - the process is logically ready to execute and has been assigned the CPU
10. XMT - the process is transmitting a message to a receiver process that is prepared to accept the message

A transition graph defining the possible interactions between these states is shown in appendix [5]. The most important point is the increase in complexity from the single processor version to the distributed version.

4.2.2 EPL Primitives

INIT - This primitive is not actually called by the EPL code, but by the startup routine. INIT takes care of whatever system initialization is necessary. The memory management abstraction sets up pointers into free memory. A free memory estimate is made for each processor in the system based on the amount available locally. For the I/O subsystem, waiting queues are set up, the data structures used to refer to the devices are initialized, and the devices are placed in a known state. An idle process is created on each processor to be run whenever the local ready list becomes empty. If the processor's I.D. is zero, the first EPL actor is also created and started running.

CREATE - A CREATE call is made by a parent actor to enter a new actor, or child, into the system. The processor with the highest local free space estimate is chosen as the site for the child. A CREATE request, containing the parent's name and the child's act, is passed to the chosen processor where a process descriptor and runtime stack are allocated for the child. The process descriptor is initialized as required, the child's state is set to NEW, and, if the child

is nonlocal, an acknowledgement with the child's name and argument count is returned. Either a dummy process descriptor containing the returned information or the actual child process descriptor (depending on whether or not the child and parent are on the same processor) is linked to the parent's process descriptor for later reference.

RUN - The RUN primitive is called by a parent actor to pass parameters to the newly created child. The number of parameters the child expects is retrieved and used to insure that no extra arguments are sent. A run request consisting of the arguments and the child's name is sent to the child's processor. After the request is received, the child's state is changed to RDY and the child is scheduled on the local ready list.

TERMINATE - This kernel call is made by an actor as it completes the code for its act. The actor is simply removed from the local ready list and its state changed to END. Neither the actor's process descriptor nor its stack space is reused.

SEND - A SEND call is made by an actor to transmit a message to another process. The sender is put into SND state and removed from the local ready list. The sender's kernel then passes an enquiry to the receiver's kernel indicating that it is prepared to send. On the receiver's processor, the sender's process descriptor, or a dummy process descriptor if it is nonlocal, is added to a linked list attached to the receiver's process descriptor for later reference. When the receiver is ready to accept the message, an acknowledgement containing the message length is passed back. The message is then sent and the sending actor returned to the ready list.

RECEIVE - An actor needing information to continue execution makes a RECEIVE kernel call. The process descriptors (or dummy process descriptors) for actors prepared to send to this receiver will be in a linked list attached to the receiver's process descriptor. If the receiver is able to accept a message from any source, then one of the possible senders is removed from the list. If the receiver has specified a particular sender,then the list is scanned and if a match is found, it is taken from the list. In either case, the receiver passes a message

request, containing the message size, to the sender. The sender responds by transmitting the message. If a sender is not available, the receiver is placed in BLK state.

SYSTEM - SYSTEM is for miscellaneous operations which are not included among the major primitives. Currently, SYSTEM calls are provided to handle reading from and writing to terminals, and voluntary actor preemption.

4.2.3 Vertical Communications

The kernel can communicate vertically with the EPL actors and with the I/O subsystem. Information passed between the actors and the kernel via the general purpose registers and the actor's run time stack. EPL actors use a trap to enter the kernel.

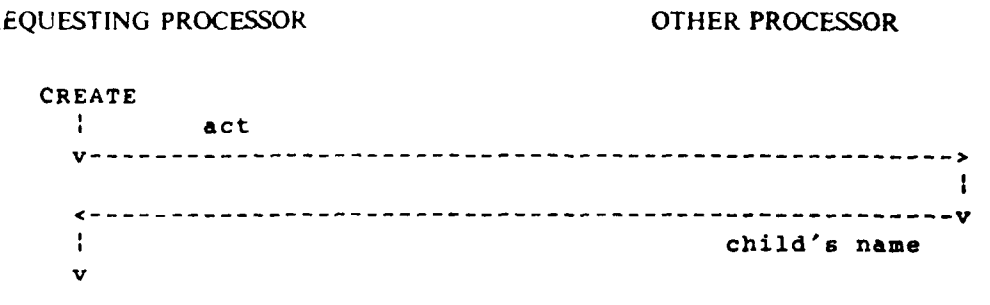
The kernel calls the I/O subsystem when it needs to communicate with another kernel to complete an EPL kernel call. All of the information necessary for the I/O subsystem to format and pass the message is placed in the process descriptor of the EPL actor.

When the I/O subsystem receives a message from another processor, it either puts the information received into a dummy process descriptor or directly into an actor's stack. In either case, the kernel is called to take whatever actions may be necessary.

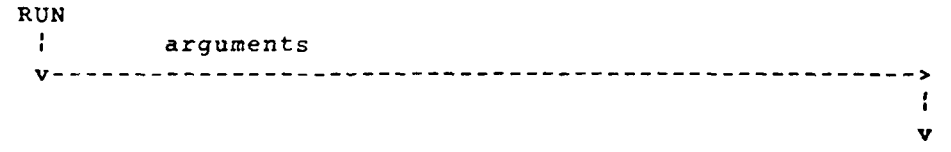
4.2.4 Horizontal Communications

The kernel may need to request either information, service or both from another kernel in order to complete a primitive. For example, Kernel A may have chosen kernel B's processor as the site for a new actor. Kernel A must pass a request for the creation of the child to kernel B, and kernel B must return the name of the child.

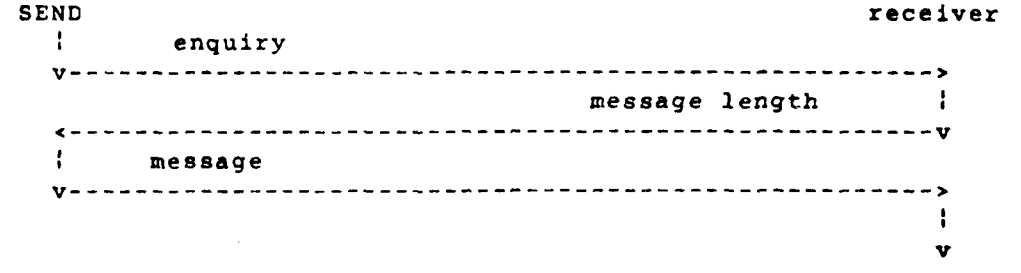
The kernel to kernel protocol is dependent on the primitive being executed. The initial protocol design was kept as simple as possible. Below is a graph representation [Stutzman] and verbal explanation of the protocol for each primitive involving kernel to kernel communication.



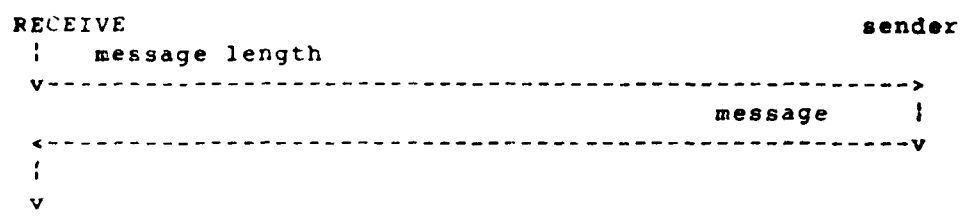
CREATE: A create request is passed form the parent's kernel to the kernel on which the new actor has been scheduled. The child's kernel returns an acknowledgement, containing the new actor's name, as well as the number of parameters it expects.



RUN: A request to run a newly created actor, along with its initialization parameters are passed from the parent's kernel to the child's. There is no acknowledgement.



SEND: The sender's kernel passes an enquiry to the receiver's kernel asking if the receiver is prepared to receive. When the receiver is ready to receive from that sender, its kernel sends an acknowledgement containing the message length to the sender's kernel. The sender's message is then passed to the receiver.



RECV: This protocol is a subset of the SEND protocol, starting with the acknowledgement of the enquiry by the receiver's kernel and ending with the arrival of the message.

The message formats are in appendix [4]. The memory layout for the kernel and the contents of different files is in appendix [3].

4.2.5 Local Policies

Multi-tasking is done from the ready list with a round robin algorithm. The running actor is preempted when it makes a kernel call if there is a ready actor to take its place. There is no protection against an actor written without kernel calls hogging the processor forever once it starts running. However, for well intentioned users there is a voluntary preemption call.

In all of the kernel functions, special concern was given to keeping memory management as simple as possible. The kernel allocates space dynamically in two ways: 1) sixteen word pieces for use as process descriptors 2) variable length pieces for use as EPL run time stacks. Once space is allocated for an actor, it is never deallocated. Dummy process descriptors, which are used by the kernel to hold information pertaining to nonlocal actors, can be reused.

4.2.6 Global Policies

The high level scheduler is designed to maximize parallelism of actors. It was decided that one way to do this was to assure a distribution of actors by allocating children nonlocally. Additional requirements were to keep the algorithm simple, avoid the use of polling, and maintain an equal load balance (if possible).

When each kernel is initialized, it sets up a local estimate of free space for every kernel in the system. Since the code for every kernel is almost identical, each assumes that all other processors have the same amount of free space as it does. On a CREATE call, the processor with the highest estimate is chosen as the location for the new actor. The local free space estimate for the chosen processor and that node's own local estimate are updated, but not any of the other processor's estimates. This means that the only correct free space estimate for a processor will be on that node and all others are too high. The net effect is a preference for placing children nonlocally, which will tend to distribute actors to all processors.

4.3 I/O SUBSYSTEM

The I/O subsystem contains the link protocol and link dependent software. The link software was isolated for several reasons. It is shared by many other modules. Most of the kernel routines which implement the EPL primitives require inter-kernel communication and need to use the inter-processor links. The separation makes the kernel independent of the interconnection protocol and hardware. This makes modifications in the interconnection schemes much easier.

The I/O subsystem is responsible for routing all kernel to kernel communications. For simplicity, the kernel is designed to be ignorant of the actor locations. It knows only that the actor making the kernel call is local. If an actor other than the one making the call needs to be referenced or manipulated then the kernel simply passes a request to the I/O subsystem to communicate with the actor's kernel. In fact, it could actually be requesting communication with itself. For example, actor A wants to SEND to actor B. A's kernel must ask B's kernel if actor B is ready to receive a message. Kernel A does not check if kernel B is itself but just asks the I/O subsystem to communicate with kernel B. If kernel B is indeed kernel A, the I/O subsystem sets up the request in the same manner as those received from nonlocal kernels and makes the appropriate kernel call.

The I/O subsystem is divided into three major sections. There is a high level transmitter which handles nearly all of the link level protocol and is independent of the link hardware. The link hardware information is in the low level transmitter and receiver which manage the actual sending and receiving of information. The receiving processor's part in the link protocol is handled by the low level receiver.

4.3.1 Vertical Communications

The I/O subsystem communicates vertically with the kernel. When the kernel requires the services of the I/O subsystem, it places all the information necessary for the call into the process descriptor of the actor involved and then calls the I/O subsystem. A kernel to kernel message received by the I/O subsystem is placed in either a dummy process descriptor, or directly into an actor's stack. The kernel is then called to perform any manipulation required for that type of message.

4.3.2 Horizontal Communications

The lowlevel link protocol is based on the concept of a connection. A request for a connection is passed from the I/O subsystem wishing to communicate to the I/O subsystem with which a connection is desired. The I/O subsystem initiating the communications then waits until the other I/O subsystem sends an acknowledgement. Once a connection is made, the processors are dedicated to the information exchange and remain connected until all the information has been passed.

Because the basic protocol requires the initiating I/O subsystem to wait for an acknowledgement from the cooperating I/O subsystem, it is possible for deadlocks to occur. Another problem is that it is possible for each of two I/O subsystems to simultaneously attempt to initiate a connection with the other. To prevent deadlocks and arbitrate contention, the processors are arranged in a hierarchy according to their ID numbers. Lower numbers have higher priority. As with any hierarchical scheme, there is the danger of starving the low priority processors. Further work needs to be done to find out how costly the problem is, and

what are the solutions. One possibility would be to distribute the actors in a way that minimizes contention.

The basic connection protocol has been extended to work with the hierarchy of I/O subsystems. A request for a connection is passed from the I/O subsystem initiating the communications to the I/O subsystem it wishes to communicate with. The initiating I/O subsystem then waits for an acknowledgement before proceeding. If it is a positive acknowledgement the message is sent. If it is a negative acknowledgement, indicating that the cooperating I/O subsystem is trying to initiate communications on the same link, its handling depends on the relative positions of the two I/O subsystems in the hierarchy. The kernel with the lower priority relinquishes the line, sends a positive acknowledgement and then waits to receive the message. It tries again later to make the connection.

While a kernel is attempting to make a connection without priority, it must listen to I/O subsystems above it in the hierarchy in order to prevent deadlocking. If one of these wishes to make a connection, the low priority I/O subsystem sends a positive acknowledgement and receives the message before returning to its own communication. Once it has made the connection, it no longer listens to the other I/O subsystems.

When a communication is received, it is handled immediately by the kernel. It is quite possible that the handling of the message will involve communications with another kernel. However, the message in question may have been received while the kernel was attempting a transmission. A flag is used to prevent nesting of transmissions, and requests made to the transmitter while it is busy are placed on a queue. The transmitter is not exited until the queue is empty.

The actual transmission and receipt of information across the links is done using wait loop I/O. The request for a connection can be detected either with an interrupt or by testing the ready bit of the link device. An interrupt can occur only when either an EPL actor or the idle actor has control of the processor. Execution within the kernel and I/O subsystem must be

protected from interrupts.

Wait loop I/O is being used as the basic method of information passing instead of interrupt I/O for several reasons. It was calculated that with five fully connected processors in the system, interruputs could occur as frequently as every six instructions. There is no special purpose hardware in the system to manage the links, and this rate is too fast for the processor to handle. The use of wait loop I/O also simplifies the taking of performance measurements. The researcher has precise control over how and when inter-processor communication will occur, enabling him/her to more easily design experiments and measure its preformance.

CHAPTER V

KERNEL PERFORMANCE MEASUREMENTS

The performance of a distributed system is a tradeoff between the benefits of increased parallelism within the distributed software and the costs incurred by the increased complexity of the kernel and the increased communications overhead. The amount of parallelism possible is influenced by many factors, such as the design of the distributed software, the local and global scheduling policies, and the availability of resources. Because the focus of this project is on the kernel, and not on scheduling policies or performance of EPL programs, only the additional costs due to distribution are to be examined. A later project will present a more complete performance analysis.

The most important factors to be considered in estimating the additional costs due to distribution are the increases in logical complexity and communications overhead. These costs are contained within the kernel and I/O subsystem. An estimate of the increase has been obtained through a comparison of the distributed kernel with an EPL kernel designed for a single processor (both written in C). The increase in logical complexity is demonstrated by comparing the two versions on the number of actor states and transitions possible and on the number of kernel entry points. The increased costs in the kernel functions were estimated by counting the number of machine instructions needed to implement the two versions of the kernel. The increased communications costs were estimated by observing the frequency of interprocessor communication, contention and preemption, and by computing the average connection waits. The wait times for setting up a connection and actually passing the message were measured by counting the number of executions of the I/O wait loop on the sending side. All measurements were taken with the test programs on three processors (appendix [5]).

5.1 LOGICAL COMPLEXITY

The logical complexity of a particular kernel implementation is dependent on its design. This makes a precise measurement of the necessary increases in complexity from the single processor kernel to the distributed kernel impossible. However, the general trend can be shown by a comparison of the two kernel implementations.

Any increase in logical complexity of the kernel can cause a corresponding increase in the implementation costs. The functions provided by the kernel are basic, and the costs of implementing them are not exorbitant. However, any increase in the implementation costs of the primitives can have a substantial effect on system performance because of the frequency of their occurrence. Appendix [7] shows a histogram of the number of actor instructions executed between primitives. In general, the observed lengths of instruction sequences between kernel calls were very short and exhibited little variation.

There is an increase in logical complexity which is reflected in actor management and in the implementation of the EPL primitives. The state diagram for an EPL actor in the single processor version (figure [5]) has six states and five transitions. A similar graph for the distributed kernel (appendix [5]) has twelve states and twenty transitions. Several of these differences are due to the partitioning of the primitives into separate functions which may be executed on different machines. For example, the CREATE primitive is divided into three parts: 1) the parent's kernel schedules the new actor on a processor, 2) that processor (possibly different than the parent's) allocates resources to the child and returns its name, 3) the parent's kernel receives the name and puts it in the parent's data segment. Most of the new states and transitions are necessary to handle the intermediate states which happen when the local part of a primitive is completed but the nonlocal part is not. In the above example, the parent cannot be allowed to run until the child's name has been returned and so between steps 1 and 3 the parent is placed in the intermediate state CRT. The remainder of the differences are caused by the use of the actor's data segment to buffer arguments to be transmitted. The SEND and RUN

primitives transmit information directly from the calling actor's data segment. During the time lapse between the kernel call and the actual data transmission, the actor cannot be run because it may alter the data.

The kernel for the single processor has only six entry points corresponding to the six primitives. The distributed version has an additional six which are entered from the I/O subsystem. The additional entry points in the distributed kernel are simply to catch responses from the nonlocal parts of a primitive. The original CREATE has essentially three entry points: the first (called create) is entered from the parent actor, the second and third (create request and child name) are entered from the I/O subsystem.

The costs of the major primitives have been estimated as the average number of machine instructions executed by the kernel implementation. Figure [6] shows the instruction counts for both the single processor kernel and the multiprocessor kernel. The count for the single processor kernel is an estimate obtained by studying the assembled code. It does not include register saving or context switching. The multiprocessor counts are a summation of the average number of instructions executed by each kernel function participating in the primitive. These functions are not necessarily executed by the same kernel, but may be done by kernel's on other processors. The counts were obtained at runtime by incrementing a counter between each machine instruction. All instructions not part of the EPL actor were considered part of the primitive. These include the saving and restoring of registers on subroutine calls, support functions, and context switching.

	single processor	distributed
CREATE	33	348
RUN	39	267
SEND	40	212
RECEIVE	50	253

Figure 6. MACHINE INSTRUCTION COUNTS FOR MAJOR PRIMITIVES

As can be seen from figure [6], the differences between the counts for the two kernels are substantial. At least part of the increase can be accounted for by the fact that the distributed kernel was designed for logical simplicity and not for efficiency. For example, the distributed kernel made much greater use of subroutines than the single processor version. No attempt has been made to estimate how much of the increase is due to the different design philosophies. The counts for the multiprocessor kernel include context switching and register saving but the single processor counts do not.

5.2 COMMUNICATIONS

The increased costs of communications can be seen on both the logical and physical levels. In the single processor kernel, the only horizontal communication is between EPL processes and it is implemented as copying within a local store. The only vertical communication is between EPL actors and the kernel, and it is handled the same way as in the distributed kernel. The distributed kernel must also support vertical communications between the kernel and I/O subsystem. Horizontal communication can occur between processors and it is sent across serial links. Transmission across serial links is clearly more expensive than local memory references.

The communication costs for a particular program are dependent on the amount of communication required and the expense of that communication. EPL processes tend to exchange many short messages and, as mentioned in the previous paragraph, make many kernel calls. In the distributed kernel, both of these may involve interprocessor communication. The communicating actors may not reside on the same processor and the primitive requested in a kernel call may require cooperation with another kernel. The amount of communication which takes place across the links is dependent on the global scheduling policy and it can be high. Even local communications are more expensive on the multiprocessor kernel. The kernel does not distinguish between local and nonlocal communications. This means that the distributed kernel can not take some of the short cuts available to the single processor kernel.

The expense of nonlocal communication depends on the amount of data passed, the cost of actually passing data on the link, and the cost of the I/O subsystem protocol. On the average, interprocessor messages are short. Kernel to kernel messages have an average length of 3.3 bytes (appendix [1]) and EPL actor communications (SEND and RUN) also tend to be short [BALK79]. As explained in chapter IV, before communication takes place between two processors, a connection must be set up. The processor wishing to send passes a request on the link and waits for a response. After the response is received, both processors are dedicated to the passing of the message. Because of this, the short message lengths, and the fact that LSI11s are not especially fast processors, the cost of actually passing data on the links is not that great. A large part of the communications costs are in the setting up of connections.

The processors have been arranged in a hierarchy to prevent deadlocking. If there is contention for a particular link, it is resolved in favor of the processor with the higher priority. A low priority processor waiting to communicate with a higher priority processor is forced to listen to all processors with priorities higher than itself. If one of these processors wishes to communicate, the low priority processor must accept the message. This is referred to as preemption and is used to break multi-processor ties. The mechanisms used to arbitrate contention and multiprocessor ties introduce both additional costs to the expense of making a connection and the possibility of starving the lower priority processors.

Some preliminary measurements of the I/O subsystem are shown in figure [7]. The transmissions in each processor were grouped into categories based on type. Transmissions can occur locally, nonlocally with priority, and nonlocally without priority. Since priority is based on kernel ID number (lower numbers have higher priorities), nonlocal transmissions from kernel zero always have priority and those from kernel two never have priority. Only kernel one has all possible types of transmissions. The nonlocal categories are broken down into subtypes. Nonlocal transmission with priority can occur with and without contention. Transmissions without priority can occur with contention and with preemption. For each of these groups, the table has both the average count of wait loops executed by the sender while setting up a

connection, and the relative frequency of each type. The nonlocal without priority group is treated slightly differently than the others. Because it has no priority, the transmission can be interfered with several times. The average wait count for all nonpriority transmissions is given along with their percentage of occurrence. Following this, are the average wait counts due to preemption and contention, and their frequency of occurrence. These last two figures are computed in relation to the total number and cost of nonpriority transmissions and not the total number of transmissions.

		MROBIN		SIEVE	
processor ID	type	avg wait count	percent occurrence	avg wait count	percent occurrance
0	local	0	20	0	0
	nonlocal w priority				
	w/o contention	153	60	130	84
	w contention	89	20	94	16
1	local	0	21	0	4
	nonlocal w priority				
	w/o contention	116	26	110	55
	w contention	94	10	91	12
	nonlocal w/o priority	147	43	115	29
	w contention	117	29	116	01
	w preemption	0	0	0	0
2	local	0	18	0	0
	nonlocal w/o priority	214	82	136	100
	w contention	113	25	110	22
	w preemption	140	50	114	42

Figure 7. I/O MEASUREMENTS

A majority of the transmissions are nonlocal. For the sieve, several processors have no local transmissions, and for mrobin the highest proportion of local transmissions is 21%. These

figures are dependent on the global scheduling policy, which did not attempt to maximize for local communications (section [4 2 6]) but for parallelism.

Nonlocal with priority transmissions are cheaper with contention than without contention. The fact that there is contention for the link means that the lower priority processor has already passed a connection request on the link and is listening for an acknowledgement. Since the lower priority processor is already prepared for communication, and not executing an uninterruptable section of code, the high priority processor can make its connection more quickly.

The average wait count for receiving the data for an entire message (not including the I/O subsystem protocol) averages approximately 100 (ranging from 78 - 144). The connection cost is generally higher than the cost of sending the actual data.

5.3 SUMMARY

The kernel for the distributed system is more complex logically than the kernel for the single processor system. Although impossible to measure precisely, this increase is evident in the approximately two to one ratio of number of actor states, actor transitions, and kernel entry points between the multiprocessor and single processor versions. Also, the implementation of the multiprocessor kernel required a substantially greater number of machine instructions than the single processor version. This increase is partly due to the emphasis on logical simplicity in the design of the multiprocessor kernel and to differences in the methods used to count the instructions.

Communications costs for the multiprocessor kernel are much higher than those for the single processor kernel. The design of the multiprocessor kernel not only required more communications, but communications were more expensive. The costs of passing information across links are greater than direct memory references. Even local communications in the distributed kernel are more expensive because the design philosophy of logical simplicity eliminated many shortcuts. Because inter-kernel and inter-actor communications are generally

quite short, the costs of implementing the I/O subsystem protocol is often greater than the costs of actually passing the information. Also, the protocol introduces the possibility of starving the low priority processors. The measurements presented in this chapter provide evidence of a general trend of increased costs, where they occur and why. It is hoped that most of these additional costs will be offset by the increased parallelism possible in the multiprocessor environment.

Note that much of the I/O overhead would be absorbed by the hardware in a more intelligent device. In any decentralized implementation it would still be necessary to resolve contention for the device, but more appropriate hardware would lessen processor time spent on I/O. Further work needs to be done to characterize the I/O required and the response of hardware interconnection structures to such loads.

CHAPTER VI

CONCLUSIONS

The major goal of the project was to gain experience in the design and implementation of an operating system kernel for a distributed system. Current research in distributed systems and languages was studied to isolate the important issues and to examine different solutions. An operating system kernel to support a distributed programming language was designed and implemented on loosely coupled, directly connected processors. Some performance measurements of the kernel were taken to approximate the increased costs due to distribution. Benefits from increased parallelism were not taken into account because the amount of parallelism is dependent on many factors other than the design and implementation of the kernel.

6.1 SYSTEMS AND LANGUAGES

A set of central concepts underlying distributed processing was derived from the research projects examined. Each project defines a process model in which the process acts as the basic unit of distribution and decentralization. Processes cooperate to accomplish a task with mechanisms for communication and synchronization. Because there is no centralized control, the systems must tolerate nondeterminism in process interactions. The projects examined were built on the concepts of processes interacting nondeterministically with communication and synchronization constructs. These concepts are implemented differently in each.

The different process models can be characterized by the process create and terminate operations and interprocess relationships. Process creation can occur either dynamically or at compile time. Processes can be either independent, autonomous entities or arranged in a strict parent- child hierarchy. Processes can be allowed to kill one another, kill themselves, or

simply terminate at code completion.

Two schemes for interprocess communication have been widely accepted: shared memory and message passing. The communication mechanism should ideally conceal the hardware configuration and network fabric. Because message passing can be used to implement communication on a shared memory machine, but shared memory cannot be implemented on a loosely connected machine, it is considered to be the more basic of the two. The message passing mechanism can provide automatic message buffering or rely on the user to buffer messages where needed. The unbuffered method requires a blocking send. Both the sender and receiver must be prepared for the communication so that the message can be moved directly between their data segments. Buffered message passing allows the sender to proceed regardless of the state of the receiver.

Interprocess synchronization can be explicit, based on an event mechanism, or implicit, based on message passing. Unbuffered message passing guarantees synchronization of the two processes involved at the time the communication occurs.

Nondeterminism between processes can be allowed by letting processes receive from one of a set of possible senders, rather than specifying a particular sender. This can be implemented as a guarded command containing either receive commands or events keyed to a message arrival.

The language chosen for the project, EPL, supports dynamically created, autonomous processes which terminate at code completion. Processes communicate and synchronize with unbuffered message passing. Nondeterminism is incorporated with a mechanism for receiving from an unspecified sender.

6.2 EXPERIENCE

6.2.1 System Information

The kernel design is based on partitioned system information. The kernel knows about

local actors only, and relies on kernel to kernel enquires to get information about nonlocal actors. Several kernel operations build queues of actors using linked lists of process descriptors. In these cases, a nonlocal actor is temporarily represented locally with a dummy process descriptor which is the same as the standard process descriptor. Information required by the operation is placed in the dummy process descriptor.

Another possible implementation would have been to replicate system information. If all process descriptors are copied locally, kernel to kernel enquiries would be unnecessary. Having information replicated could enhance recoverability. Other information could be replicated to simplify system algorithms, such as the global scheduling policy. However, maintaining replicated information can be expensive. Mechanisms must be provided for updating the replicated information and guaranteeing its integrity. These mechanisms would involve a great deal of interprocessor communications.

The partitioned approach is simple but increases the communication costs of kernel operations because of kernel to kernel enquiries. Replication offers the benefits of having useful information locally, but adds the costs of maintaining a distributed data base.

6.2.2 Memory Management

The use of unbuffered message passing greatly simplifies memory management. Messages are copied directly from the sender's data segment to the receiver's data segment. Problems of flow control need not be dealt with in an automatically buffered scheme. The kernel must handle buffer shortages. Messages for a receiver must be queued. If the message is being passed between processors, it must be buffered and queued at both locations. The case of a receiver not accepting its messages can create cleanup problems and buffer shortages. Variable length messages complicate the buffering scheme and fixed length messages could be overly restrictive.

The major disadvantage of unbuffered message passing is that it prevents the sender from running until the receiver is ready to accept the message. However, the user can overcome this

if

necessary, by creating a new process dedicated to sending the message. This secondary process is blocked instead of the parent.

6.2.3 Hardware Transparency

The system software is divided into two sections. The kernel manages the actors and provides the environment for the distributed programming language. The I/O subsystem manages the processor links and routes kernel requests to the correct processor. All kernel requests for information about an actor other than the calling actor are passed to the I/O subsystem for routing. If the actor is local, the request is passed back up to the kernel, if not, it is sent the proper kernel. Requests originating on other processors look identical to those originating locally.

With this design the kernel does not distinguish between local and nonlocal actors and does not know whether a request originated locally or nonlocally. The physical distribution of actors is transparent to the kernel, simplifying its design.

6.2.4 EPL Simplicity

EPL was designed as a system's programming language. It provides a minimal set of primitives intended to be building blocks for more complex structures. The simplicity of EPL is reflected in the simplicity of the kernel design.

6.3 MEASUREMENTS

The increased costs, but not the benefits, from distribution were estimated. A comparison of two kernel implementations, the multiprocessor version and one for a single processor, shows substantial increases in logical complexity. The multiprocessor kernel has twice as many entry points and states, and four times as many state transitions. The greater complexity is reflected in the number of assembler instructions needed to implement the EPL primitives. Both versions are implemented in C . The comparison is intended to show a trend of

increased costs and their characteristics, and not intended to be a precise measurement.

The costs of communication were measured as the number of wait loop executions per interprocessor communication. It was found that, in many cases, the link protocol cost more than the actual passing of information.

6.4 IMPROVEMENTS

6.4.1 Actor Naming

In this kernel design, process names are unprotected from user errors. The EPL actors store and manage the actual actor names. The names are used for interprocess communication. The kernel does not check to ensure valid names. Each actor must have a unique, global name. A terminated actor's name cannot be reused because that could result in a process mistakenly communicating with a later actor of the same name. The kernel uses the address of an actor's process descriptor as its name. The requirement that names not be reused means that process descriptors for terminated actors cannot be allocated to new actors. This places limits on the system's dynamic functioning.

6.4.2 Global Scheduling Policy

The goal of the global scheduling policy was to maximize parallelism. The algorithm used tries to achieve this by favoring the placement of children on a different processor than the parent. In system testing the processor with the first actor was much more heavily loaded than the others, and some processors were hardly used. The imbalance in processor loads lessens parallelism.

6.5 SIMPLICITY

The philosophy of simplicity enabled the kernel to be designed cleanly and concisely. Memory management is minimal, the physical distribution is hidden from all but the lowest level, and the kernel supports only a rudimentary set of primitives.

6.6 FURTHER WORK

Further work needs to be done to understand the system's characteristics, to evaluate the current design in light of these characteristics, and to make alterations to improve performance. In particular, the communication load needs further study. The current I/O structure should be evaluated and tailored to that type of load. The effect of scheduling on performance needs to be examined. The scheduling mechanism can then be optimized. Is the architecture suitable? Perhaps a different architecture would be more appropriate.

The effectiveness of EPL as a language for distributed processing needs to be evaluated. It may be that EPL cannot be used to achieve the goals of distributed processing or that it needs some changes before it can be used as the basis for a system.

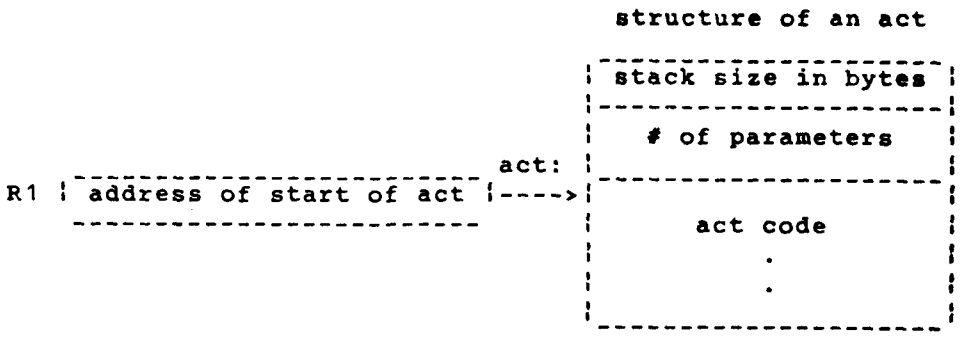
APPENDIX I

EPL TO KERNEL INFORMATION PASSING

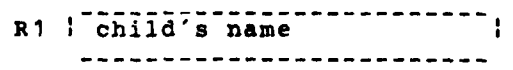
unless followed by dots, a slot is one word

CREATE

INFORMATION PASSED

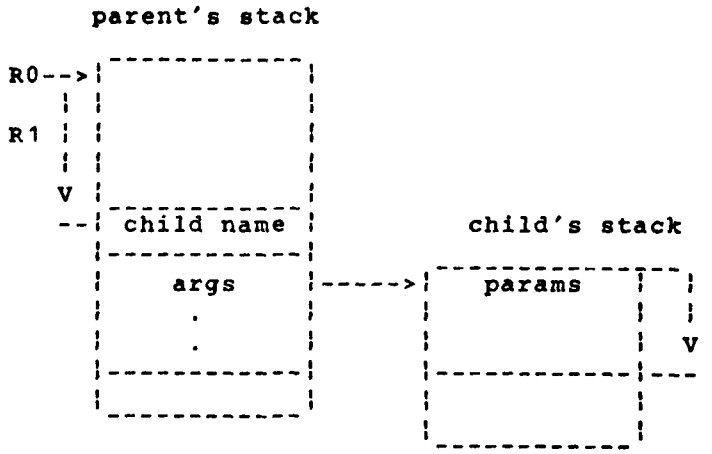


INFORMATION RETURNED

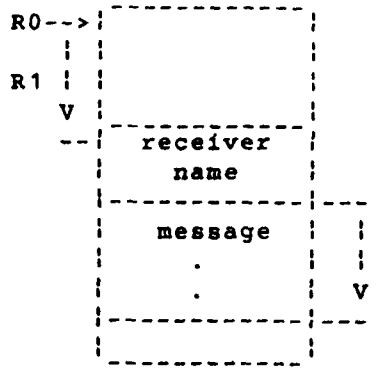


RUN

INFORMATION PASSED

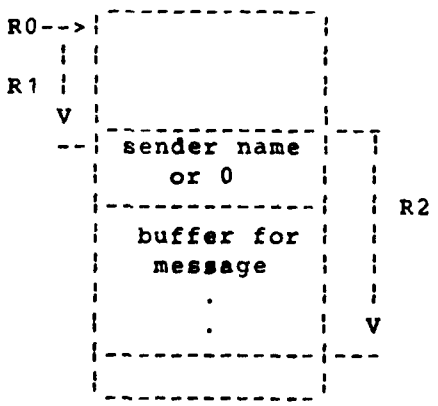


SEND
INFORMATION PASSED

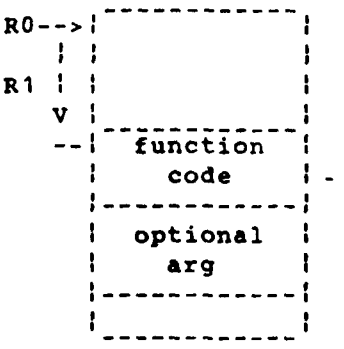


msg. length known by receiver

RECEIVE
INFORMATION PASSED

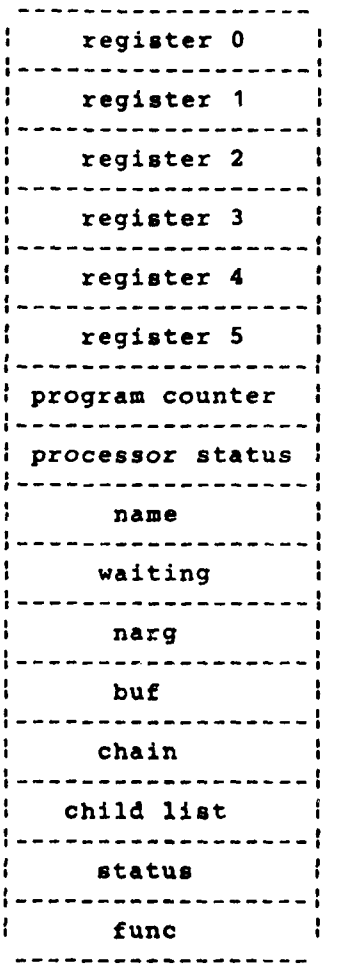


SYSTEM
INFORMATION PASSED



APPENDIX 2

ACTOR PROCESS DESCRIPTOR



name

Name is used to pass information to the I/O subsystem. It holds the name of an actor which is to be the subject of a kernel to kernel communication. The lowest three bits of the name indicate the kernel with which the communication is to take place (could be the local kernel).

waiting

Waiting contains a pointer to the start of a list of process descriptors for actors waiting to send to this actor. It is used by the SEND/RECEIVE primitive.

narg

Narg is used to hold the number of arguments to be sent for both child initialization (CREATE/RUN) and inter-process message passing (SEND/RECEIVE).

buf

Buf contains a pointer into the actor's stack. It is the sum of the contents of R0 (stack pointer) and R1 (offset into stack) which are set up by the EPL actor for the primitives RUN, SEND/RECEIVE and SYSTEM.

chain

Chain is used to connect the process descriptor into a linked list. It holds either a pointer to the next process descriptor or nil if it is the end of the list. Since an actor can only be in one list at a time (ready list, child list, waiting list) chain is used by all the lists.

child list

Child list contains a pointer to the start of a list of process descriptors for children created by the actor but not yet started running. It is used by CREATE/RUN.

status

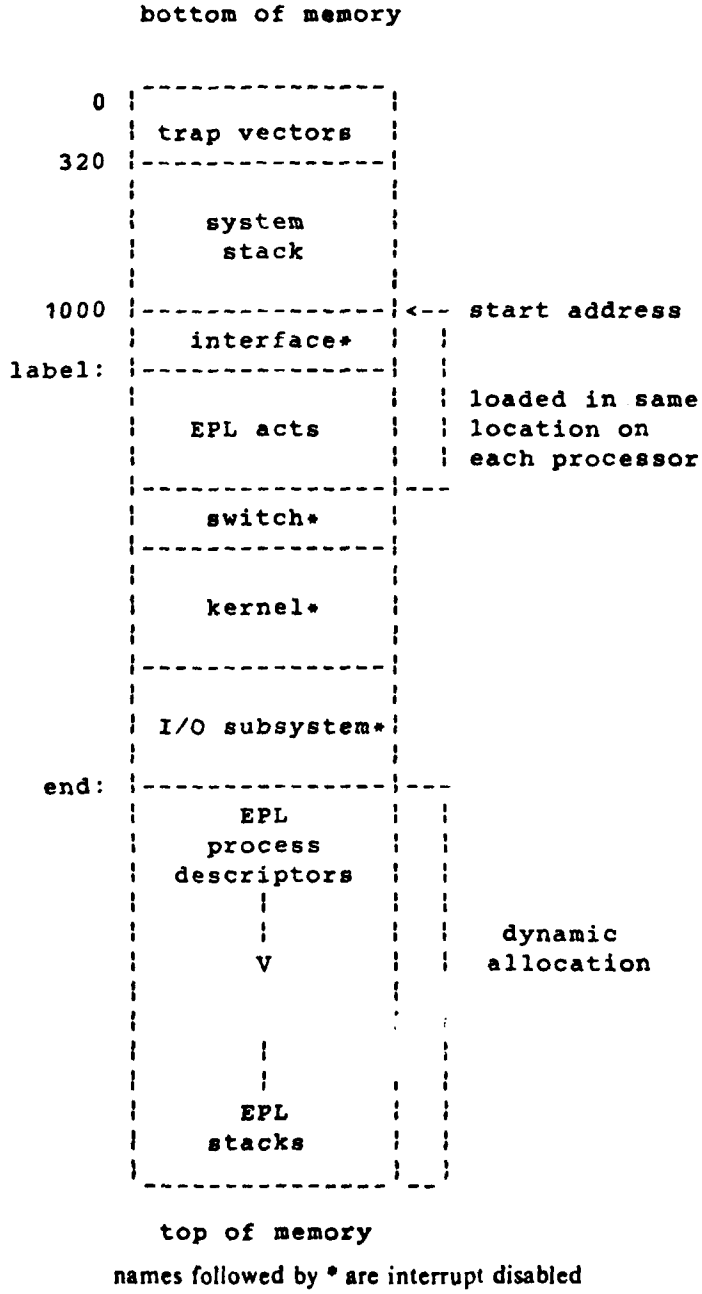
Status holds an integer value representing the current state (as indicated by the actor transition graph, FIGURE []) of the actor.

func

Func is used to pass information to the I/O subsystem. It holds an integer which informs the I/O subsystem what type of message is to be passed.

APPENDIX 3

MEMORY LAYOUT AND FILE DESCRIPTION



names followed by * are interrupt disabled

interface

Written in Unix assembler, the interface provides the bridge between the the EPL and the kernel (written in C) environments, and contains all of the assembler code needed for certain operations such as I/O. When the kernel is entered from an EPL actor the interface saves the general purpose registers, program counter, and processor status register in the running actor's process descriptor. Upon exit, this information is restored from a possibly new running actor . The interface contains all of the trap handlers (I/O interrupts, break point trap, kernel trap), the code for the idle actor, and the software multiplication and division routines used by EPL (no longer necessary because the hardware is available). The Unix assembler used does not include several standard LSI instructions, such as HALT or MTPS and MFPS (to change processor status). The machine code for no operand instructions, such as HALT have been inserted with direct assignments. MTPS and MFPS are simulated by placing the new processor status and program counter on the system stack and then executing a return from trap (RTT). The code for bringing up the kernel starts at location 1000 within the interface . It initializes the system stack, processor status register, calls the kernel initialize routine and dispatches the first actor (the first EPL actor if the kernel ID is zero and the idle actor otherwise).

label

Label is an assembler file containing a label to mark the start of the EPL code.

kernel

The kernel functions have been grouped into four files. Kernel.c contains the functions called directly by EPL actors (also called the nonpreemptive functions). Prempt.c has kernel functions called from the I/O subsystem. These are referred to as preemptive functions because the I/O subsystem can be entered by means of an interrupt, thus preempting an EPL actor. Auxfns.c contains the auxiliary functions, such as memory allocation, and scheduling. All of the linked list manipulating routines are in llist.c.

I/O subsystem

The I/O subsystem is divided into three main groups. The high level transmitter (xmt.c) handles all of the link level protocol. The low level transmitter (xmtih.c) manages the actual sending of messages and the low level receiver (rcvih.c) handles the receipt of messages.

end

End is a variable assigned by C to the end of the loaded code.

APPENDIX 4

FORMATS FOR KERNEL TO KERNEL MESSAGES

unless otherwise specified, slots are one word

create request

Receipt of a create request message causes the kernel to create a new actor locally. The name of the parent is passed so that the kernel knows where to return the child's name. The name of the act is the start address of its code. (part of CREATE)

name of act
name of parent

child name

The child name message is used to return the name of the child to the parent actor. The number of initialization arguments required by the child is also included (part of CREATE).

name of child
number of arguments expected (1 byte)
name of parent

run request

A run request message contains the initialization arguments for a newly created actor. Once the child is initialized it can be started running. The number of arguments being passed is included in the message so that the receiver handler does not need to refer to the child's process descriptor for the information. It is not really essential. (part of RUN)

name of child
number of arguments that follow (1 byte)
arguments

enquiry

The sending actor's kernel passes an enquiry message to the receiver's kernel to indicate that a sender is prepared to send a message. (part of SEND/RECEIVE)

name of sender
name of receiver

acknowledge enquiry

The acknowledge enquiry message is the response to an enquiry message. It is passed from the receiving actor's kernel to the sending actor's kernel when the receiver is prepared to accept the message. The sender's name is passed so that the sender's kernel can locate it. The The number of arguments to be sent, information known by the receiver, is needed by the sender's kernel to complete the message passing. (part of SEND/RECEIVE)

name of sender
number of arguments + 1 expected by the receiver (1 byte)

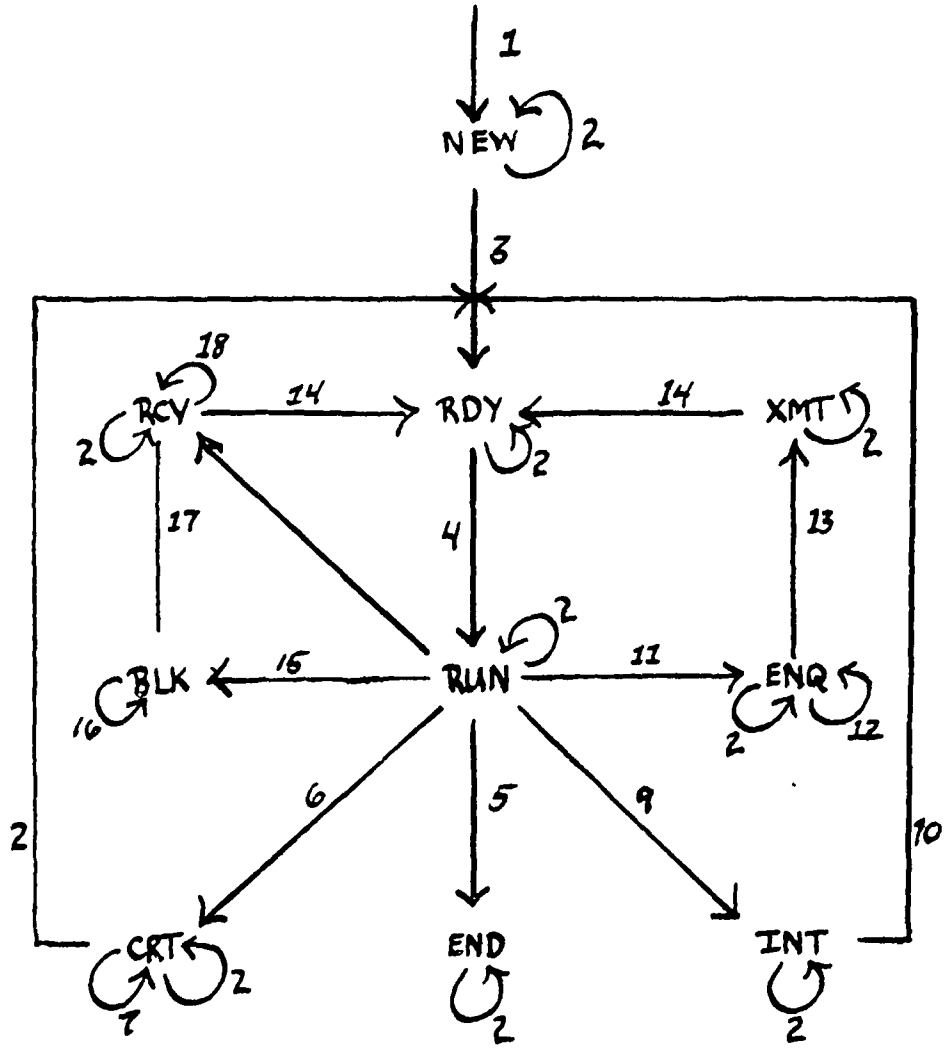
message received

This message contains the message expected by a receiver. (part of SEND/RECEIVE)

receiver's name
arguments

APPENDIX 5

THE DISTRIBUTED PROCESS STATE TRANSITION GRAPH



THE PROCESS STATE DEFINITIONS FOR THE DISTRIBUTED KERNEL

BLK: the process is waiting to receive a message and a suitable sender process has not yet enquired if the process is prepared to receive

CRT: the process is creating a child process and has not yet received the name of the child process

END: the process has terminated (the process may be referred to by other processes)

ENQ: the process has sent a message to enquire if the receiver process is prepared to accept a message

INT: the process is transmitting parameters to a child process

NEW: the process has been created (i.e., it has a name), but not parameters have been transmitted to it by its parent process

RCV: the process has acknowledged a sender process and is waiting to receive the text of the message

RDY: the process is logically ready to execute, but has not been assigned the CPU

RUN: the process is logically ready to execute and has been assigned the CPU

XMT: the process is transmitting a message to a receiver process that is prepared to accept the message

THE TRANSITIONS

1: a parent process requests the creation of a child process

2: a sender process enquires if the process is willing to receive a message

3: the parameters needed to initialize the child process have been received

4: the process is assigned the CPU resource

5: the process terminates

6: the process initiates the creation of a child process

7: the request to create a process has been transmitted

8: the name of the child process has been returned

9: the parent process wishes to transmit parameters initializing a child process

10: the parameters have been transmitted to the child process

11: the process wishes to send a message to a receiver process, an enquiry is sent to the receiver process

12: the enquiry has been transmitted

13: the receiver process has acknowledged that it is prepared to receive the message

14: the message has been transmitted

15: the process wishes to receive a message from a sender process; no acceptable sender has enquired if the process is ready to receive

16: an unacceptable sender enquires if the process is ready to receive a message

17: an acceptable sender enquires if the process is ready to receive a message

18: the acknowledgement has been transmitted

19: the message has been received

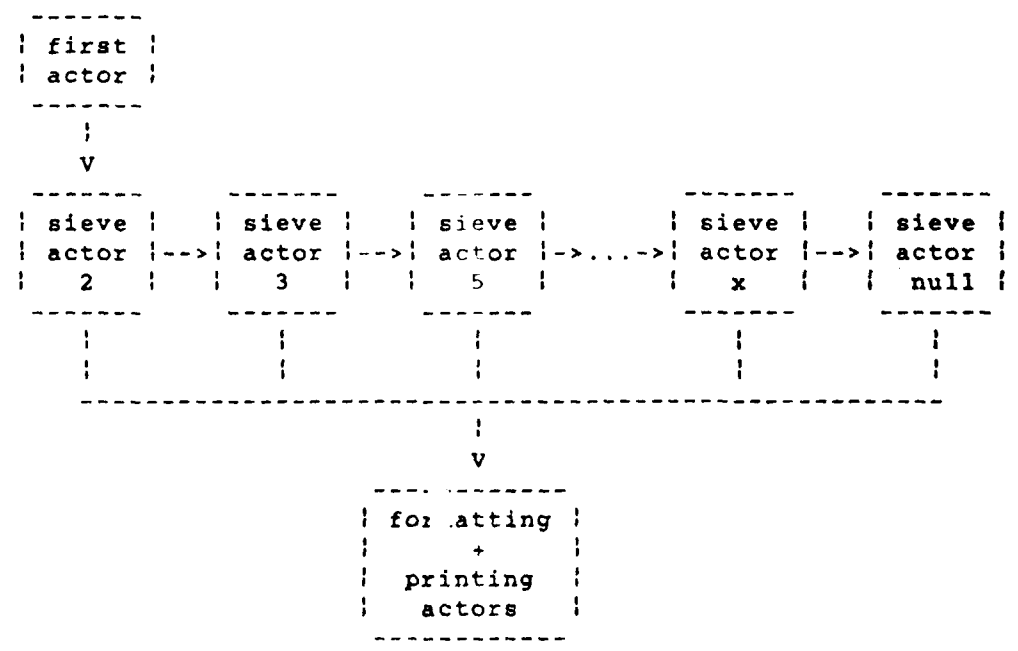
20: the process wishes to receive a message from a sender process; a acceptable sender has already enquired if th process is ready to receive

APPENDIX 6

TEST PROGRAMS

Sieve

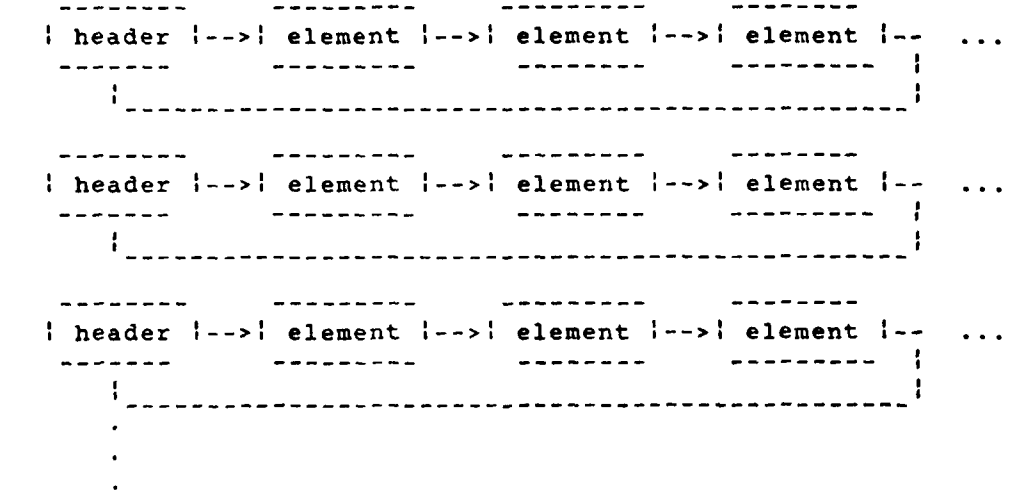
The sieve of Eratosthenes is implemented as a chain of actors. The first actor generates possible prime numbers with a counter which are passed down the chain via the SEND/RECEIVE mechanism. Each of the remaining actors (sieve actors) possesses a previously generated prime number which it uses to filter multiples from the set of possible primes. At the end of the chain is a sieve actor waiting to receive the next prime. After it receives a prime number, it creates a new sieve actor to wait for the following prime. The new prime is then sent to a set of formatting and printing actors and the sieve actor becomes part of the filter.



Mrobin

Mrobin sets up a structure, similar to a two dimensional array, in which each element is an actor. There are three types of actors involved: element actors are elements in a row, header actors each create a row, and the initial actor (only one is created) creates header actors. The header actor creates new element actors, and adds them to the head of the row. Each row is a circular linked list of elements. After the header creates a new element, it prints the new actor's name and starts a null message down the row chain. When the message returns to the header from the last element, the next element is created. As each element receives the signal,

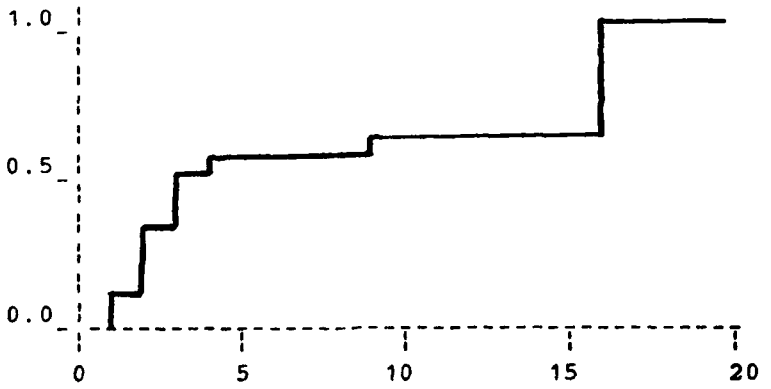
it sends the name of the next actor in line to a printing actor, and then waits for the next signal. With this arrangement, there is no parallelism allowed within a row.



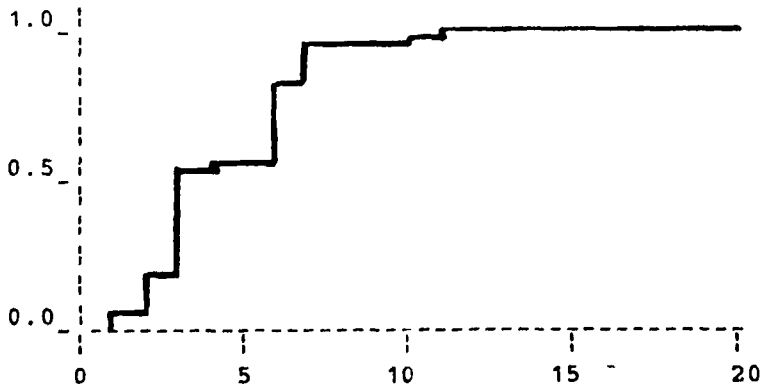
APPENDIX 7

DISTRIBUTION OF LENGTHS OF INSTRUCTION SEQUENCES

MROBIN



SIEVE



Part III

Design of an Operating System for Distributed Communicating Processes

by

J. Morse

1. INTRODUCTION

This paper reports on an attempt to apply some of the basic principles of software engineering to the design of an operating system for multiple loosely-coupled mini-computers. The goal of the project was to develop a design that was independent of the CPU on which the operating system was to run, and on the communicatons facility available to link the CPU's. Such a design should make it possible to implement a "family" of operating systems [Par76]. Possible variations among the family members include:

1. Implementation for several different interconnect facilities with varying performance parameters.
2. Implementation for various network topologies, including in the limit the single CPU case.
3. Incorporation of new operating system features such as time dependent scheduling.

The design approach was to identify a set of requirements, express those requirements in abstract terms, and then refine the abstractions to formulate a design. In refining the requirements into a design, principles of software engineering were used that were drawn from two areas in which a great deal of progress has been made in the last ten years. The first of these areas is modularization of design and implementation with the goal of easing the expansion, contraction, and subsetting of large software systems. Much of the work in this area has been done by Parnas -- the bibliography gives numerous references. The second area is the use of data abstraction as both a design tool and as a principle to guide implementation. In abstraction for design, the contributions of Flon [Flon75] and Guttag [Gutt80] should be mentioned. In the development of programming languages based on data abstraction, prime examples are CLU [Lisk77], ALPHARD [Wulf76], and EUCLID [Pope77].

The operating system supports the programming language EPL [May79]. EPL is based on the principle of communicating processes very much like CSP [Hoare78] or the Distributed Processes of [Brin78]. Processes are autonomous; they communicate and synchronize with each other only through message passing. EPL was chosen as the basis for the study of operating system design because it requires only a small number of primitive operations:

* Send a message
* Receive a message
* Create a process
* Run (initiate) a process

A number of other distributed operating systems were considered, including HYDRA [Wulf74], Cm* [Joes77], Pilot [Red80], and Medusa [Oust80]. An operating system to support EPL was chosen over these other possibilities because its simplicity allowed the design process to focus on the essential problem of distributed control of sequential communicating processes without the need to deal with peripheral issues such as memory management and file systems.

2. REQUIREMENTS

2.1 Abstract Requirements

Analysis of the definition of the EPL programming language suggests that there are two basic abstractions that the operating system requires -- actors, and messages. For the remainder of this discussion, I shall use the more conventional term "process" in place of the EPL term "actor". The requirements for an operating system to support EPL may be expressed as follows:

1. There shall be Processes and Messages

2. Processes can do the following with reference to other Processes:
   * Create a Process
   * Run (Initiate execution of) another Process
   * Send a Message to another Process
   * Receive a Message from another Process
3. A Process may terminate itself.
4. Each Process may make progress independently of any other Process except while executing:
   * Send
   * Receive
5. The result of Receive will depend soley on the state of the corresponding Sending Process at the time that the Send is executed.
6. Create and Run have no effect on the state of the Process executing them.
7. Apart from the above, Processes can be modelled as independent state machines.

The first three requirements listed above follow directly from the specification of EPL. Requirements 4-7 are an attempt to refine the notion of Processes and the operations that they can perform in an axiomatic way. These "axioms" conform to the usual notions about what a "process" is. They form the basis for a set of invariants that must be preserved by all valid designs and implementations of any family member.

The given set of axioms implies an equivalence between message passing and synchronization of processes. Yet Parnas in [Par79] cites combining message passing with synchronization as an example where components perform more than one function, thus making it difficult to implement a subset that provides one feature without the other. In spite of Parnas's warning, I will accept for this design that the semantics of EPL have inexorably bound synchronization with message passing, and that the design of the operating system will be based on the same principle. If it were required to provide a mechanism for synchronization in the

absence of message passing, then some other mechanism, such as semaphores, would be required for synchronization. The added complexity does not seem warranted in the present case.

2.2 Refinement of the Requirements

We now work from the abstract requirements toward a set of concrete requirements, by introducing new abstractions. This may be viewed as a process of describing increasingly lower level "virtual machines" [Par79], or as a process of identifying levels of data abstraction.

The requirement that processes may make independent progress and may be regarded as independent state machines suggests that the concept of process splits into 3 sub-concepts: 1) each process has a "current state", 2) each process has a set of rules for making transitions from state to state, and 3) there is some kind of executor which causes processes to make state transitions. We will refine these three notions in terms of, respectively, 1) process descriptors, 2) executable statements, and 3) the operating system dispatcher.

The process descriptor is a basic abstraction of the design. There is a one-to-one correspondence between process descriptors and processes. All state associated with a particular process is either contained in, or referenced by, its process descriptor. Various modules will need to know about various components of the process descriptor. A basic criterion for partitioning the system is to restrict the use of each component to the minimum number of modules.

Executable statements define what a process does. We assume that these statements form a list of instructions that is re-entrant, and so shareable by multiple processes. The term used in the EPL definition for such a statement list is "ACT". One of the items that must be kept in the process descriptor as part of the state information is a pointer to the "current location counter" for this list of instructions.

The dispatcher is the module that allows processes to make progress. The operating system scheduler in effect multiplexes the CPU by repeated calls to the dispatcher to advance the state of a particular process. The requirement of independent progress requires that the scheduler will eventually submit every process that is able to make progress to the dispatcher. We may add a further requirement that all processes ready to run get "fair" treatment in the sense that once a process has received service, it will receive no further service until all other ready processes have been dispatched as least once. A fundamental goal of an operating system is to keep the CPU busy advancing the state of runable processes whenever possible.

2.3 Extension to Multiprocessor Implementations

Up to this point, nothing has been said about supporting multiple processors. If we assume that processes may reside on different CPU's in different physical locations, what effect will this have on the requirements?

The existence of multiple CPU's clearly requires that we have multiple schedulers, since it is the scheduler that provides the multiplexing of a single CPU among multiple processes. Since the scheduler is at the top of the requirements hierarchy, this suggests (not too surprisingly) that each CPU needs a copy of the entire operating system. This operating system will have the same requirements as outlined above, with the additional requirement that the scheduler be able to deal with process descriptors and queues on remote processes.

To incorporate remote processes into the operating system, it is only necessary to provide a communications facility so that remote process descriptors and queues can be accessed. Presumably, the same data abstractions will apply to remote data as to local data. An additional requirement is that the access routines for process descriptors and queues be able to distinguish local objects from remote objects. The dispatcher and other support modules need no change whatsoever. A pos-

sible difficulty is in the Create Process operation, which needs some notion of creating a process at a remote CPU.

With the proposed approach, the details of the communication facility, and knowledge about the system topology, are all isolated to the communications module. The goal with this approach is to allow for changes in both communications facilities and system topologies with no change to any part of the operating system except the communications module. As we shall see, this approach does not work.

3. DESIGN

The boundary between requirements specification and design specification is not hard and fast. The second and third parts of the previous section began to talk about abstractions such as "process descriptor" and modules such as "scheduler" and "dispatcher" which might be considered part of design specification. However, the refinements made in the last section were almost inevitable given the fundamental requirements of EPL. On the other hand, in this section refinements represent choices from among many alternatives.

3.1 Module Definition

The criteria for dividing the operating system into moduls will be based on those given in [Par79].

A. Information will not be distributed. Parnas talks elsewhere [Par72] about using "information hiding" as a criterion for modularization. The design will attempt to achieve information hiding by the mechanisms of data abstraction. Because of the complexity of an operating system, the data abstractions will be hierarchical. As an example, the process descriptor is a data abstraction. Many modules need to deal with process descriptors, but most need know in detail about only a small part

of the whole. This will be achieved by defining the process descriptor as an abstraction which is itself made up of abstractions. Some modules need to know only that the descriptor exists, without knowing any of its detail. Other lower level level modules need to know about the existence of some of the sub-abstractions without knowing how to manipulate them. Finally, the lowest level module, the one that implements a realization of the abstraction, needs to deal with the exact details.

B. The design will avoid the data transformation model. Modules will deal with the data abstractions as they are, without changing formats. This is almost an automatic result of the data abstraction approach, since changes to data are only done through the operations associated with the abstraction.

C. Modules will perform single functions.

D. Loops will be avoided in the "uses" relation. Parnas provides a definition of the "uses" relation based on the dependence between modules. This design will incorporate the notion of dependence on data abstractions in the "uses" relation. This is to say that module A "uses" module "B" if module A makes use of a data abstraction provided by module B.

3.2 Abstract Data Types

It follows almost directly from the requirements specification that we will require process descriptors and messages. In addition, the operating system will need to be able to organize process descriptors for easy access as needed on a one-by-one basis. It will also need to manipulate messages in a similar way. A message may be sent to a given process before that process is ready to receive. The scheduler will need a way to save the message and keep track of it until the process signals that it is ready to receive. To handle both process descriptors and messages, the queue abstraction will serve.

3.3 Notation

A variant of Pascal is the design language, augmented with the abstract type definition notation of [Flon75]. Some extensions and changes to Pascal and Flon's notation have been adopted for convenience:

1. Each top-level name in the "DECLARE" part of a type definition that is preceded by the keyword "EXTERN" is visible to the modules which use the type.
2. Fetch and store operations are implicitly associated with each such external name of the type, so that one may write, for instance, "new.state = old.state". This eliminates the necessity to define an abundance of operators which merely fetch and store values.
3. Enumeration type declarations create named constants with the same scope, so we may write "new.state = Wrecv". This does not seem to violate the goal of information hiding; if constants of this type were not available, an externally visible operator would be required to accomplish the same purpose.
4. DECLARE is used instead of the Pascal VAR, to declare a pointer to a typed variable. Except in the create operation of a TYPE, no implicit creation is done; the variable is initialized only by assignment. All variables are pointers, as in CLU [Lisk76].

This choice of a design language presents a compromise between precision on the one hand, and wordiness on the other. With the extensions cited above, it is possible to convey all the details of the data abstraction approach required, without getting into the details of representations. The language is wordy enough to convey its semantic meaning to the casual reader. At the same time it is concise enough to represent the essential details of the design of an operating system in just a few pages of "pseudo-code".

3.4 The Initial Design

The initial attempt to design the operating system is given in Appendix I. The design implements the basic operations Create, Run, Send,

AD-A099 195 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
DECENTRALIZED SYSTEMS.(U)
DEC 80 E E BALKOVICH
UNCLASSIFIED TR-CS-15-80 DAS660-79-C-0117 NL

2 OF 3
AD A 099195

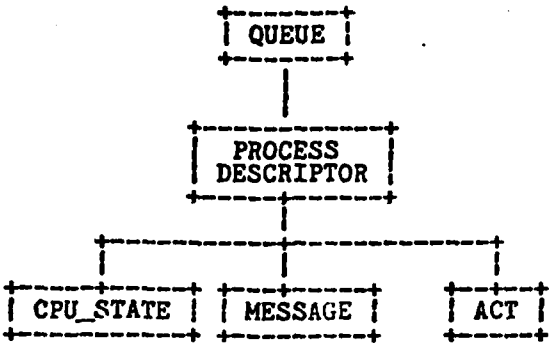
and Receive, where Receive is a general receive -- the process will accept a message from any other process. The top level of the scheduler is a loop which repeatedly gets the current process from the ready queue, dispatches the process to the CPU, and upon return, executes the system call that caused the process to return to the scheduler.

Execution of system calls is done by the procedures in the module "sc_procs". These procedures maintain the state of the system as a whole by moving process descriptors between the ready queue, which contains descriptors for processes that are able to run, and the idle queue, which contains descriptors for processes that are blocked waiting to send or receive messages.

The data abstractions to support the operating system are basically those that represent process descriptors, queues, messages, and the physical CPU. This partitioning seems appropriate from a modularization point of view since it neatly separates four critical aspects of the operating system design:

1. Definition of Processes
2. The Queueing discipline and implementation
3. Message formats
4. Hardware details

One of the underlying principles of the design is that the data abstractions form a hierarchy, with lower level abstractions being used to form higher level abstractions. The following diagram illustrates the relationship among some of the major data abstractions used in the design. It also represents the "uses" relations among the data abstractions.



Even though many of the details at the lower levels are omitted from this design, it gives an adequate framework for the analysis of the operating system, and appears to have achieved the goals or the current project. Its limitations will be discussed in the follwing section.

4. CRITIQUE

The previous section optimistically proclaimed the truth and beauty of a proposed operating system design, and claimed that it had achieved some goals of information hiding and modularity. This section will examine the faults of the design, and in particular how some software engineering techniques, even applied informally, can predict the problems with this design.

4.1 Circularities

Messages point to process descriptors; process descriptors point to messages. This interdependence between processes and messages points up the fact that the operating system is using messages to implement process synchronization.

System commands (send, receive, create, run) are the "undefined terms" that both the lowest level (the user process) and the highest

level (the scheduler) deal with. This is a symptom of an essential co-routine relationship between the operating system and user processes. For the purpose of the design, the scheduler is placed at the top of the control hierarchy -- it appears to call user processes as closed subroutines. But on the other hand, the user process "calls" the operating system to receive some service such as message transmission or process creation. There is a two-way flow of stimuli in an operating system -- the system issues "proceed" stimuli to processes; processes issue "syscall" stimuli to the operating system. In the current design, the "proceed" stimulus is treated as a normal call operation, while the "syscall" stimulus is implemented through a back-door return parameter from the process. This is a compromise solution because the existing abstractions do not adequately deal with a co-routine control structure.

Of these two major loops in the "uses" relation between abstract data types, the message/process descriptor seems minor. Interestingly enough, it would be possible to implement the scheduler in such a way that messages were never pointed to by process descriptors, though at some cost of convenience. On the other hand, messages will certainly have to reference process descriptors, since they are used for communication and synchronization between processes. This suggests that the message is the higher level concept in the design.

I feel that the loop in the "uses" relation of system commands, on the other hand, is both significant and unavoidable. There is no obvious way of eliminating it, nor of re-arranging the levels in the hierarchy of abstractions to make the control structure cleaner.

4.2 Consistency

Investigation of the design in terms of the axioms of the requirements section must study how process descriptors are manipulated in the system. The responsibility to insure progress of all processes (requirement 4.) suggests as an invariant of the operating system that every process descriptor be on exactly one queue. The only queues that

the scheduler has are the two called ready, and idle. During execution a process may alter its own process descriptor in order to set up system calls. But the processes have no access to the system queues. The only place where process descriptors and queues are manipulated is in the modules called system, and sc_procs. The following is an informal argument for the consistency of the process descriptor management.

1. The module sc_procs is the only place where information of any kind is passed between process descriptors.
2. The top level scheduler always removes the current process from the ready queue, dispatches it, and then calls exactly one procedure in sc_procs.
3. Every procedure in sc_procs, under all path conditions, inserts the current process descriptor into exactly one queue.
4. In every path in every procedure in sc_procs, whenever a process descriptor is successfully removed from a queue, it is subsequently inserted into exactly one queue.
5. Therefore, the desired invariant holds at the beginning of the WHILE loop in the Scheduler.

4.3 Multiple CPU's?

The above consistency discussion points up the fact that the desired invariant holds at the top of the WHILE loop. In fact, the invariant is a pre-condition for the subsquent execution of the call to the dispatcher, which is the only place where the system actually "makes progress" (from the process point of view). Consider the problem of maintaining the invariant in procedure sc_send if the destination process is on another processor. The process descriptor referenced by variable p is accessible only by some communications link. Also, the idle and ready queues for p are remote and are accessible only via the link. Then the following 4 statements:

```
IF p.state = Wrecv THEN BEGIN      /* test state */
   p.msg := m;                     /* transfer message */
```

```
q.remove (idle, p);          /* remove from idle queue */
q.put_end (ready, p);        /* insert in ready queue */
```

each require communication. The design can be improved substantially by combining the last 3 statements into one operation that transfers the message and then allows the operating system at the remote site to move p from the idle to the ready queue. Nevertheless, the delay for communications has been inserted into the scheduler at a point where the desired invariant does not hold. Thus the dispatcher will not be called, and no process can progress while the communication is going on. This violates the independent progress axiom since a process may be ready to run, the CPU is available (it is just waiting for communications to complete) yet no process can make progress.

The problem is that the operating system has failed to perform its basic function -- to keep the CPU busy. To achieve this, it must insure that all the operations it performs directly are CPU bound. Any waiting that is done, even on behalf of system level processing, must allow the main scheduler loop to run, thereby keeping the CPU busy. In fact, only processes can wait; the function of the operating system is to provide a means whereby processes can wait without holding up the CPU. The obvious conclusion is that communications must be handled at the process level.

5. THE COMMUNICATIONS PROCESSES

At this point the design has been forced back to an approach that I had considered earlier and discarded on the grounds that it did not seem "clean". That approach is to handle all inter-CPU communications through a pair of processes which handle the network link. For the purpose of design exposition, these two processes will be treated as if they were normal user processes, even though one of them is actually an interrupt handler. An efficient implementation of this approach would

probably give these processes special status, preferred scheduling, special CPU context, and so forth, in order to service the link in a timely manner. These two processes are called start_io and io_done.

5.1 Process Design

Whenever the scheduler encounters a Send to a remote process, it performs the normal send processing as if the message were sent to start_io. Process start_io, which is created and initiated at system startup time and never terminates, simply loops on a Receive primitive. When it receives a message, it starts the link hardware to send the message to the appropriate remote CPU. It then waits for a message from io_done signalling that the message has been sent and the link is available for the next message.

Process io_done is really an interrupt handler. For a clean conceptual model, assume that the scheduler in fact dispatches it periodically to poll the done status of the link. If the link is done, either a message initiated by start_io has finished, or a message from another CPU has been received. In the first case, io_done sends a "done" message to start_io to allow it to get the next outbound message. In the second case, io_done determines which local process the message is destined for, and does a Send to that process.

5.2 Interaction with the Operating System

A little reflection will show that the proposed solution will provide the last 3 of the 4 steps involved in executing a Send to a remote process. The message will be transfered, and, as a result of io_done doing a local Send to the destination process, the process descriptor of the destination process will be moved from the idle queue to the ready queue.

The original code depended on the pre-condition that the destination process was already in the Wrecv state, and so was guaranteed to

already be on the idle queue. It may be that the Send occurs before the Recv. One approach to handling the remote synchronization is to design a moderately elaborate handshaking protocol that defers the message transfer until both processes are in the appropriate state. This approach is in fact used in an existing operating system for EPL. A conceptually simpler approach, however, is to always let Sends proceed immediately. If the scheduler gets a message from a remote process via io_done, and the destination process is not ready to receive it, it will create a temporary process descriptor for the remote process, save the message in it, and place it on the idle queue. This allows the scheduler to place io_done back on the ready queue immediately, so that link service is not held up. When the destination process is ready to receive, the scheduler must make sure that the temporary process descriptor is discarded -- if it is mistakenly placed on the ready queue, disaster will strike in very short order.

5.3 Implications of the Approach

The new design for the scheduler and its modules is shown in Appendix II. Note that all the changes have been made at the top level -- none of the data abstractions have been changed in any way. Two details have been omitted. One is that Create and Run of remote processes has not been designed. There needs to be a mechanism for returning an identifier of the remote process to the creating process. The second detail is that in the single CPU design, process descriptors are used by the scheduler as identifiers for the processes they represent. For the multiple CPU design, there need to be identifiers for remote processes that can be distinguished from the process descriptors for local processes.

Reviewing the requirements in light of the new design yields an interesting phenomenon. Requirement 5 is in fact still met, but by a chain of indirection. A message from source to destination passes from source to start_io (local CPU), to io_done (remote CPU), possibly to a dummy process descriptor, and finally to the destination process. Since requirement 5 is met for each of these steps, it is met for the opera-

tion as a whole. Synchronization, however, is definitely not achieved. The original sending process can continue as soon as the message has been transferred to start_io, and long before the message gets to the destination. The new design effectively buffers message transfer, and message sending can no longer be used for real-time synchronization even though it still achieves logical synchronization.

6. SUMMARY

This attempt to design an operating system using principles of software engineering has been a qualified success. Certainly the discipline of applying the data abstraction approach has kept the design cleaner, and clearer, than might otherwise have been the case. Progress has been painfully slow, and many design approaches that at first seemed promising led to either unworkable or impractical solutions.

It was hoped that the approach would be able to address the issues of operating system families. The design is at a stage now where it should be easy to show how other CPU's, various communications facilities, and new features can be accomodated. Both the writer's energy and the reader's patience are certainly exhausted by now, so the demonstration of the level of modularity that has been achieved will wait for another day.

The most important design principle used in this exercise was that of keeping careful control over access to information. A strong distinction emerged between access to the name of a data item and access to its internal structure. By placing the names of (references to) abstractions into the structure of higher level abstractions, a hierarchy of data abstractions was developed which served to keep the design both coherent and modular.

APPENDIX I

```
/* Module Definitions for the Operating System                    */

/* The scheduler is the top level module of the Operating System.
   It makes use of the process descriptor, message, and queue
   abstractions to manage the CPU.  It assumes that The Force
   has incarnated an act called MainAct which is initially
   made ready, and which creates and runs all the other processes
   of the system.                                                  */

MODULE system =

PROCEDURE scheduler =
DECLARE
    current: proc_des,                      /* current process      */
    ready:   queue (proc_des),              /* queue of ready processes */
    idle:    queue (proc_des);              /* queue of idle processes  */
BEGIN
    ready := queue.create (proc_des);       /* the ready queue      */
    idle := queue.create (proc_des);        /* the idle queue       */
    current := proc_des.create (MainAct);   /* the current process  */
    queue.put_end (ready, current);         /* start with main process */

    WHILE not queue.empty (ready) DO BEGIN  /* run so long as some process
        current := queue.get_next (ready);  /* get next process to run */
        CASE proc_des.dispatch (current);   /* run process, test command */

            Create:     sc_create (current.new_act);
            Run:        sc_run (current.new_proc);
            Send:       sc_send (current.smsg);
            Recv:       sc_recv ()

            END of CASE
        END of WHILE
    END of scheduler
END of MODULE system.
```

```
/* Definition of sub-procedures to execute each command          */
/* Single CPU version                                             */

MODULE sc_procs =
/* These procedures import the following from the surrounding
   scheduler context:                                             */
IMPORT
    current: proc_des,                      /* current process    */
    ready:   queue (proc_des),              /* ready queue        */
    idle:    queue (proc_dex);              /* idle queue         */

PROCEDURE sc_create (a: act) = BEGIN        /* Create a new process */
    current.new_proc := proc_des.create (a); /* new process descriptor */
    queue.put_end (idle, current.new_proc); /* put onto idle queue */
    queue.put_end (ready, current)          /* re-enable current proc. */
    END of sc_create;

PROCEDURE sc_run (p: proc_des) = BEGIN      /* Run an idle process */
     IF (queue.remove (idle, p)) not = NIL  /* if on idle queue,  */
        THEN queue.put_end (ready, p);      /* make it ready      */
     queue.put_end (ready, current)         /* re-enable current proc. */
     END of sc_run;

PROCEDURE sc_send (m: msg) =                /* send a message */
DECLARE p: proc_des;
BEGIN
    p := m.dest;                            /* destination process */
    IF p.state = Wrecv THEN BEGIN           /* if ready to receive */
        p.msg := m;                         /* transfer the message */
        q.remove (idle, p);                 /* remove dest from idle */
        q.put_end (ready, p);               /* make destination ready */
        q.put_end (ready, current)          /* make current ready  */
        END
    ELSE BEGIN                              /* not ready to receive */
        current.msg := m;                   /* save message       */
        current.state := Wsend;             /* waiting to send    */
        q.put_end (idle, current)           /* suspend current    */
```

```
        END
    END of sc_send;

PROCEDURE sc_recv =                          /* receive a message */
DECLARE p: proc_des;
BEGIN
    p := find_msg (idle, current)            /* find a process trying to
    IF p not = NIL THEN BEGIN                /* if there is one      */
        current.msg := p.msg;                /* transfer the message */
        q.put_end (ready, p);                /* enable sending process */
        q.put_end (ready, current)           /* enable current       */
        END
    ELSE BEGIN
        current.state := Wrecv;              /* waiting to receive   */
        q.put_end (idle, current)            /* current now idle     */
        END
    END of sc_recv
END of MODULE sc_procs.
```

```
/* Abstract Data Types for the Operating System                    */

/* The process descriptor is the abstraction that embodies the
   notion of independent processes, modelled as state machines.
   The process descriptor also contains abstractions that allow
   processes to communicate with each other by sending messages.   */

TYPE proc_des =
     DECLARE
        EXTERN: state: {Wsend, Wrecv},
        EXTERN: context: cpu_state,          /* CPU context        */
        EXTERN: msg: msg,                    /* message to be sent,*/
                                             /* or just received   */
        EXTERN: new_act: act,                /* act for create     */
        EXTERN: new_proc: proc_des;          /* process created    */

     OP dispatch (p: proc_des): command = BEGIN
        /* Dispatch the process by giving it to the CPU to execute */
        RETURN cpu_run (p.context);
        END of OP dispatch;

     OP create (a: act): proc_des = BEGIN
        /* Create a new process descriptor */
        DECLARE p: proc_des;                 /* create new descriptor */
        p.state := Idle;                     /* initially idle     */
        p.context := cpu_state.create (a);   /* initial context    */
        p.msgq := queue.create (msg);        /* initial message queue */
        END of OP create;
END of TYPE proc_des;

/* Message abstraction.  A message has a source process, a
   destination process, and text                                   */

TYPE msg =
     DECLARE
        EXTERN: source: proc_des,            /* source process     */
        EXTERN: dest: proc_des,              /* destination process */
```

```
        EXTERN: text: string;                    /* text                  */

    OP create (s, d: proc_des, n: INTEGER): msg = BEGIN
        DECLARE m: msg;                          /* create a new message */
        m.source := s;                           /* source process       */
        m.dest := d;                             /* destination process  */
        m.text := create.string (n);             /* room for text        */
        END of OP create;
END of TYPE msg;

/* Generalized Queue.  A Queue is an ordered list of items
   which may be accessed from either end of the queue, or
   by specifying the item explicitly.                                     */

TYPE queue (t: TYPE) =
    DECLARE head, tail, quelem;        /* both head and tail point
                                          to queue elements               */

/* The operations will be declared, but not defined here.                 */

    OP get_next (q: queue (t)): t = BEGIN
        /* get_next removes and returns item from front of the queue      */
        END;

    OP remove (q: queue (t), item: t): t = BEGIN
        /* remove removes and returns item specified.  If it is
           not in the queue, returns a special constant NIL               */
        END;

    OP put_end (q: queue (t), item: t) = BEGIN
        /* put_end inserts an item at the end of the queue                */
        END;

    OP empty (q: queue (t)): Boolean = BEGIN
        /* returns true if queue is empty, else false.  */
        END;

    OP create (t: TYPE): queue = BEGIN
        /* create a queue */
```

```
        END;
END of type queue;

/* Abstract type cpu_state captures the notion of the state of the
   hardware that must be established for each process, and saved when
   the process is suspended.                                        */

/* This is the only hardware-dependent module, and would normally be
   realized in the assembly language of the CPU.  The example given
   here is for an anonymous mini-computer.                          */

TYPE cpu_state =
     DECLARE r0, r1, r2, r3, r4, r5, sp, pc, ps: INTEGER;

     OP cpu_run (c: cpu_state): command =
        /* load up the CPU registers with the state information and
           "resume" execution.  When the processes executes a system
           call, the command is returned as the result of cpu_run   */
        END of OP cpu_run;

     OP create (a: act) = BEGIN
        /* create a new state for a process to begin execution of
           the code identified as "act"                             */
        DECLARE c: cpu_state;

        c.pc := a;                          /* point to start of act */
        c.sp := sys_alloc;                  /* allocate stack space  */
        c.ps := disable;                    /* interrupts disabled   */
        END of OP create;
END of TYPE cpu_state;

/* Abstract data type "act" embodies the notion of a list of
   executable instructions (procedure body, if you will).  An
   act is created by a compiler.                                    */
```

APPENDIX II

```
/* Module Definitions for the Operating System                    */

{  Multiple CPU version - lines commented with { } are changes
   from the single CPU version.                                    }

/* The scheduler is the top level module of the Operating System.
   It makes use of the process descriptor, message, and queue
   abstractions to manage the CPU.  It assumes that The Force
   has incarnated an act called MainAct which is initially
   made ready, and which creates and runs all the other processes
   of the system.                                                  */

MODULE system =

PROCEDURE scheduler =
DECLARE
    current: proc_des,                  /* current process      */
    start_io: proc_des,                 { start_io process     }
    io_done: proc_des,                  { io_done process      }
    ready:   queue (proc_des),          /* queue of ready processes */
    idle:    queue (proc_des);          /* queue of idle processes  */
BEGIN
    ready := queue.create (proc_des);   /* the ready queue      */
    idle := queue.create (proc_des);    /* the idle queue       */
    start_io := proc_des.create (StartIO);  { create start_io  }
    queue.put_end (idle, start_io);     { make it idle         }
    io_done := proc_des.create (IODone);    { create io_done   }
    queue.put_end (ready, io_done);     { always ready to test done }
    current := proc_des.create (MainAct);   /* the current process  */
    queue.put_end (ready, current);     /* start with main process */

    WHILE not queue.empty (ready) DO BEGIN  /* run so long as some process
        current := queue.get_next (ready);  /* get next process to run */
        CASE proc_des.dispatch (current);   /* run process, test command */

            Create:     sc_create (current.new_act);
```

```
            Run:        sc_run (current.new_proc);
            Send:       sc_send (current.smsg);
            Recv:       sc_recv ()

            END of CASE
        END of WHILE
    END of scheduler
END of MODULE system.
```

```
/* Definition of sub-procedures to execute each command          */
{ Multiple CPU version                                           }

MODULE sc_procs =
/* These procedures import the following from the surrounding
   scheduler context:                                            */
IMPORT
    current: proc_des,                      /* current process    */
    start_io: proc_des,                     { start_io process   }
    io_done: proc_des,                      { io_done process    }
    ready:   queue (proc_des),              /* ready queue        */
    idle:    queue (proc_dex);              /* idle queue         */

PROCEDURE sc_create (a: act) = BEGIN        /* Create a new process */
    current.new_proc := proc_des.create (a); /* new process descriptor */
    queue.put_end (idle, current.new_proc); /* put onto idle queue */
    queue.put_end (ready, current)          /* re-enable current proc. */
    END of sc_create;

PROCEDURE sc_run (p: proc_des) = BEGIN      /* Run an idle process */
     IF (queue.remove (idle, p)) not = NIL  /* if on idle queue,   */
        THEN queue.put_end (ready, p);      /* make it ready       */
     queue.put_end (ready, current)         /* re-enable current proc. */
     END of sc_run;

PROCEDURE sc_send (m: msg) =                /* send a message */
DECLARE p: proc_des;
BEGIN
    p := m.dest;                            /* destination process */
    IF {p is remote} p := start_io;         { divert remote messages
                                              to start_io process  }
    IF p.state = Wrecv THEN BEGIN           /* if ready to receive */
        p.msg := m;                         /* transfer the message */
        q.remove (idle, p);                 /* remove dest from idle */
        q.put_end (ready, p);               /* make destination ready */
        q.put_end (ready, current)          /* make current ready  */
        END
```

```
        ELSE BEGIN                                  /* not ready to receive */
            IF current = io_done THEN BEGIN         { special for io_done   }
                p := process_des.create (dummy)     { create a dummy process }
                p.msg := m;                         { insert the message    }
                p.state := Wsend;                   { waiting to send       }
                q.put_end (idle, p);                { put on idle queue     }
                q.put_end (ready, current)          { allow io_done to continue }
                END
            ELSE BEGIN
                current.msg := m;                   /* save message         */
                current.state := Wsend;             /* waiting to send      */
                q.put_end (idle, current)           /* suspend current      */
                END
            END
        END of sc_send;

    PROCEDURE sc_recv =                             /* receive a message */
    DECLARE p: proc_des;
    BEGIN
        p := find_msg (idle, current)               /* find a process trying to
        IF p not = NIL THEN BEGIN                   /* if there is one      */
            current.msg := p.msg;                   /* transfer the message */
            IF {p not dummy}                        { if not from dummy process }
                THEN q.put_end (ready, p);          { enable sending process }
            q.put_end (ready, current)              /* enable current       */
            END
        ELSE BEGIN
            current.state := Wrecv;                 /* waiting to receive   */
            q.put_end (idle, current)               /* current now idle     */
            END
        END of sc_recv
    END of MODULE sc_procs.
```

BIBLIOGRAPHY

[Brin78] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," Comm. ACM, 21,11 (Nov. 1978), 934-941.

[Flon75] Flon, L. "Program Design with Abstract Data Types," Technical Report, Carnegie-Mellon Univ., (June 1975).

[Gutt80] Guttag, J., "Notes on Type Abstraction (Version 2)," IEEE TSE, SE-6,1 (Jan. 1980), 13-23.

[Hab76] Habermann, A. N., Flon, L., and Cooprider, L., "Modularization and Hierarchy in a Family of Operating Systems," Comm. ACM, 19,5 (May 1976), 266-272.

[Hoare78] Hoare, C. A. R., "Communicating Sequential Processes," Comm. ACM, 21,8 (Aug. 1978), 666-677.

[Jones77] Jones, A., Chansler, R., Durham, I., Feiler, P., and Schwans, K., "Software Management of Cm* -- a Distributed Multiprocessor," National Computer Conference Proceedings, NCC 1977, 657-663.

[Lamp80] Lamport, L., "The Hoare Logic of Concurrent Programs," Acta Informatica, 14,4 (June 1980), 21-37.

[Ledg77] Ledgard, H. and Taylor, R., "Two Views of Data Abstraction," Comm. ACM, 20,6 (June 1977), 382-384.

[Lisk75] Liskov, B., and Zilles, R., "Specification Techniques for Data Abstractions," IEEE TSE, SE-1,1 (Mar. 1975), 1-

[Lisk77] Liskov, B., Snyder, A., Atkinson, R. and Shaffert, C., "Abstraction Mechanisms in CLU," Comm. ACM, 20,8 (Aug. 1977), 564-576.

[May79] May, M. D., and Taylor, R. J. B., The EPL Programming Manual, Distributed Computing Project Report No. 1, Department of Computer Science, University of Warwick, Coventry, England, (1979).

[Morr80] Morris, J., "Programming by Successive Refinement of Data Abstractions," Software - Practice and Experience, 10,4 (1980), 249-263.

[Oust80] Ousterhout, J., Scelza, D., and Sindhu, P., "Medusa: An Experiment in Distributed Operating System Structure," Comm. ACM, 23,2 (Feb. 1980), 92-105.

[Par72A] Parnas, D. L., "A Technique for Software Module Specification with Examples," Comm. ACM, 15,5 (May 1972), 330-336.

[Par72B] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM, 15,12 (Dec. 1972), 1053-1058.

[Par76A] Parnas, D. L., "On the Design and Development of Program Families," IEEE TSE, SE-2,1 (March 1976), 1-9.

[Par76B] Parnas, D. L., Handzel, G., and Wurges, H., "Design and Specification of the Minimal Subset of an Operating System Family," IEEE TSE, SE-2,4 (Dec. 1976), 301-307.

[Par79] Parnas, D. L., "Designing Software for Ease of Extension and Contraction," IEEE TSE, SE-5,2 (Mar. 1979), 128-137.

[Pope77] Popek, G., Horning, J. J., Lampson, B. W., Mitchell, J., and London, R., "Notes on the Design of EUCLID," Proc. Language Design for Reliable Software, SIGPLAN Notices, 12,3 (March 1977), 11-18.

[Red80] Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., and Purcess, S., "Pilot: An Operating System for a Personal Computer," Comm. ACM, 23,2 (Feb. 1980), 81-92.

[Silb79] Silberschatz, A., "Communication and Synchronization in Distributed Systems," IEEE TSE, SE-5,6 (Nov. 1979), 542-546.

[Wulf74] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F., "HYDRA: The Kernal of a Multiprocessor Operating System," Comm. ACM, 17,6 (June 1974), 337-345.

[Wulf76] Wulf, W., Shaw, M., and London, R., "An Introduction to the Construction and Verification of APLHARD Programs," IEEE TSE, SE-2,4 (Dec. 1976), 253-265.

Part IV

Extensions to the Programming Language EPL

by

E. Balkovich

1.0 INTRODUCTION

The programming language EPL [May78, May79, Taylor80] was developed for experimental use with distributed computer systems. The language was designed and implemented for the Digital Equipment Corporation PDP-11 and LSI-11 computers as part of a distributed computing research project at the University of Warwick. It was adopted for use in this research effort after an initial collaboration between researchers at the University of Warwick and the University of Connecticut.

The language has been used in two roles by the research activity at the University of Connecticut:

(1) to define the features of a virtual machine implemented by the operating system kernel of a distributed computer system, and

(2) to provide a vehicle for exploring and evaluating the programming concepts needed to decentralize and logically distribute functions performed by software in a distributed computer system.

The use of EPL in the these roles led to several suggestions for language extensions. These extensions were largely motivated by the issues associated with exception handling and task assignment.

The purpose of this report is to document the extensions to EPL that were proposed and implemented as part of this research effort. These changes redefine the virtual machine implemented by the operating system kernel of a distributed computer system so that its features are application independent, yet sufficient to support the decentralization of application software. This document should be viewed as a specification for the prototype operating system kernel implemented as part of this research effort and future enhancements to that prototype.

1.1 TASK ASSIGNMENT

EPL can be used to write a single concurrent program whose processes can be executed by the computers of a distributed computer

system. This requires a policy for task allocation that governs which computer will be assigned to execute a particular process.

Several approaches to the problem of task allocation in distributed computer systems have been summarized in [Chu80]. The allocation of tasks is sensitive to the flow of information between processes, and thus depends on the characteristics of the application. These approaches to allocation are futher complicated by constraints imposed by fault-tolerance objectives. For example, it may be required that redundant processes not be allocted to the same computer. These considerations suggest that task allocation in real-time systems is strongly dependent on the nature of the application. Thus, a description of the desired allocation should be possible at the level of the application.

With this objective in mind, EPL was modified so that the allocation of processes to computers could be expressed as part of the application code. Initial versions of EPL deferred the allocation of processes to the virtual machine implemented by the operating system kernel supporting EPL. The language was revised so that the virtual machine assumed by EPL programs provides only application independent services related to interprocess communication and synchronization. In the new version of EPL, language features are supported that allow the application programmer to specify the assignment of processes to computers in a distributed computer system.

1.2 EXCEPTION HANDLING

A primary responsibility of the software for a distributed computer system is the management of failures and reconfigurations of the computer system. One way of achieving this objective is to use algorithms that are logically distributed [Jones80] or decentralized. Examples of such algorithms [LeLann74] generally prescribe the response required of an application to isolate and recover from a detectable fault. In order to support experimentation with decentralized software, EPL was extended to define a virtual machine that was capable of detecting of reporting

faults. These extensions allow the virtual machine concepts of EPL to be used to assess the value of these programming concepts in decentralizing software, the performance of decentralized software, and the implementation issues associated with the virtual machine.

The extensions to EPL were chosen so that the virtual machine defined by the language provides only fault detection and reporting features that are application independent. The virtual machine is assumed to make no attempt at fault isolation or recovery since these responsibilities fall in the scope of the application. It was assumed that faults detected by an operating system kernel could be mapped into failures of virtual machine operations dealing with process interactions. The feasiblity of this approach for a variety of system architectures is yet to be assessed and is the subject of future research. It has only been possible to state initial requirements for fault detection and to perform a preliminary evaluation of their implementation using the single system architecture supporting this study.

1.3 OVERVIEW OF THE REPORT

This report documents the extensions to the programming language EPL that were proposed and implemented as part of this research effort. These changes refine the definition of the virtual machine required to support applications written in EPL and thus have an impact on operating system kernel functions and their implementation.

The bulk of these language extensions were implemented in a futher collaboration between researchers from the University of Connecticut and the University of Warwick. This report is intended to serve as documentation of these changes. Section 2 provides a detailed description of the changes to the syntax and semantics of EPL. Most of these extensions deal with exeptions occuring during the interaction of processes. The actual mechanisms for detecting and reporting such errors is assumed to be the responsiblity of the operating system kernel that implements EPL. Section 3 identifies types of errors that could be detected by the

operating system kernel for EPL without elaborate hardware support. Although it appears to be possible to implement algorithms for detecting these errors using the system architecture supporting this study, the content of section 3 should be viewed as formulating initial requirements for future implementations of the operating system kernel.

Since the following sections assume that the reader is familiar with EPL, the original EPL report and the virtual machine definition are included as appendicies.

2.0 LANGUAGE EXTENSIONS

This study assumed that an EPL program would have and integrated view of a distributed computer system (i.e., an EPL program would execute on a collection of computers rather than a uni-processor). Rather than writing a collection of EPL programs, one per site, a single EPL program would be used to control a distributed system.

Figure 2-1 illustrates the levels of software required to support such a system design. EPL programs are defined by multiple processes

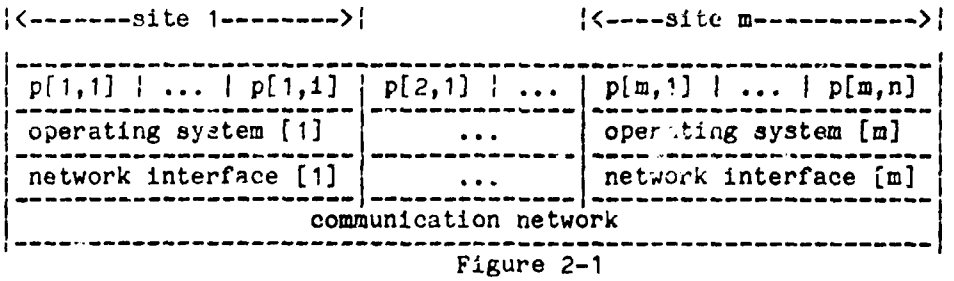


Figure 2-1

Levels of System Software

(p[1,1] ... p[m,n]). Each process is assigned to a specific site for execution when it is created. Each site requires an operating system

kernel to implement processes and their interactions. The operating system kernels require a communication network in order to coordinate their actions and to transfer data between processes. The details of a prototype of this design are givenn in [Fontaine 80].

One of the major thrusts of decentralized control is to design and build software systems that tolerate failures and/or changes in system configuration. Such algorithms must must be expressed in terms of concepts that allow a programmer to recognize and respond to faults or changes in the system. This study was specifically intertested in algorithms that tolerate the loss of a processing site, or that find alternative ways of providing functions and access to information.

The following subsections discuss extensions to the EPL programming language that are intended to be used in expressing decentralized algorithms. These changes impact both the creation of processes and interprocess communication. The extensions proposed allow a programmer to directly control assignment of newly created processes to processing sites, and to respond to detectable faults occurring when processes interact.

The following subsections will present the changes that have been made to EPL to accomplish these goals. The discussion of process creation and deletion will be separated from the discussion of interprocess communication. Each discussion will present the syntax and semantics of new concepts, and will discuss how the concepts were implemented for the LSI-11. Generally, the extensions have tried to preserve much of the flavor of EPL.

2.1 PROCESS CREATION AND DELETION

In EPL the ACTOR statement is used to generate new processes. Each process (or actor in EPL terminology) is autonomous and may be supplied with initial parameter values. In many cases, these parameter values specify other processes that will interact with the newly created process. It is this requirement that motivates the concept of a parallel

creation in which two or more processes can be created and made aware of each other through initial parameter values.

The current BNF of EPL specifies an actor declaraction as follows:

<actor declaration> ::= ACTOR <actor> { , <actor> } *

<actor> ::= <identifier> : <tertiary> :: <command>

The scope of the identifier used to name a newly created process is local to the statement containing the actor declaration statement. The actor declaration returns a value equal to the name of the newly created process(es) when used as part of an expression. Process terminate when there are no further commands to execute.

Four extensions were made to the language concepts dealing with process creation and deletion. First is the option to specify the site of creation for a new process. It is desirable, for reasons of reliability and task allocation to specify that processes be executed at specific sites of the system. This change makes an aspect of the computer system architecture visible to the EPL program. However, it tends to make the functions of the operating system kernel application independent, since there is no generally acceptable task allocation policy that can be implemented by an operating system kernel. The second extension is the option to specify a response if the operating system kernel detects a failure to correctly create and initialize a new process. Detectable failures are discussed in section 3.

The remaining extensions were motivated by frequently used programming constructions in EPL. The first of these allows a parent process to specify its name as a parameter when creating processes. The second of these allows a process to specify its termination at an arbitrary point in its execution.

2.1.1 Syntax and Semantics

The following new key words have been added to the languge: CPU,

FINISH, ONFAILURE, and SELF. FINISH defines an additional command that can be used to terminate a process at arbitrary points. SELF defines a constant whose value is the name of the process.

The syntax of actor declarations has been extended as follows:

<actor declaration> ::= ACTOR <actor> { , <actor> }* { ONFAILURE <tertiary> }

<actor> ::= <identifier> { CPU <tertiary> } : <tertiary> :: <command>

The site of each new actor may be specified separately. Any failure to create or transmit initial parameter values to a new actor will cause the tertiary following the ONFAILURE statement to be executed immediately. If an ONFAILURE tertiary is not specified on a fault is detected the default response is to terminate the creating actor.

Figure 2-2 illustrates the semantics of this statement. It shows a number of sequential create operations used to allocate space for instances of simulaneously created processes. Following that are an equal number of run operations that transmit initial parameter values to these newly created processes and subsequently enable them for execution. The failure of any of these operations transfers control the the code for the tertiary following the ONFAILURE statement. If this option is not specified and a failure occurs, a FINISH command is executed. If all operations are successful, or if the tertiary following the ONFAILURE does not terminate the process or redirect control flow (i.e, LOOP or BREAK), then the command following the ACTOR declaration is executed.

2.1.2 Implementation

The LSI-11 implementation of these concepts continues to use the following the operating system kernel functions to implement the creation of new processes:

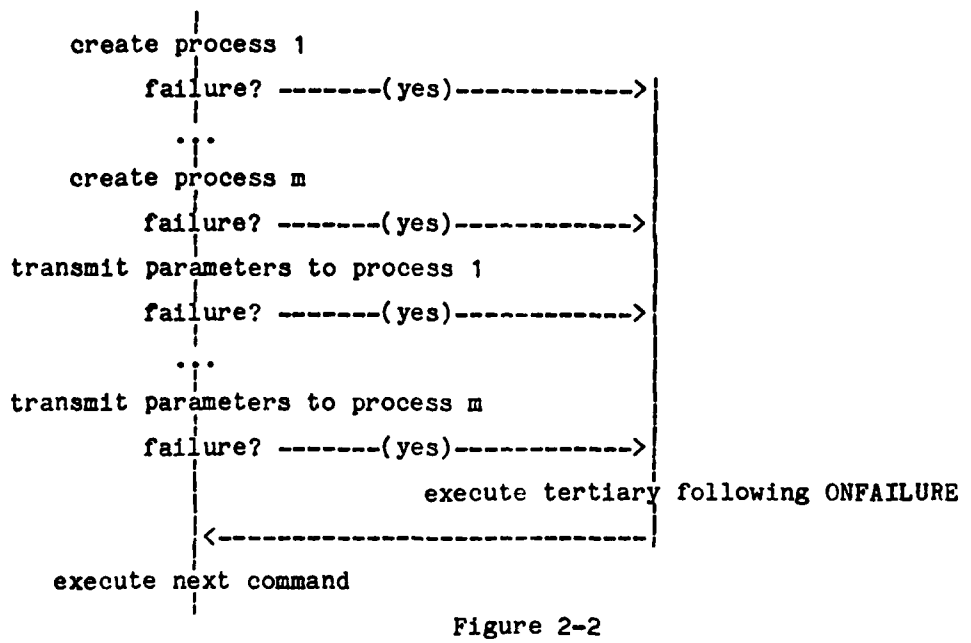


Figure 2-2

Semantics for Process Creation

screate: allocate space for and return the name of a new process

srun: transmit initial parameters to a new process and make it ready to execute

The LSI-11 implementation of these functions has modified the format and interpretation of parameters that are passed between a process and the operating system kernel. Each process continues to use a single data segment whose length constant and known at the time the process is created. The first word of the segment is assumed to be initialized by the operating system kernel with the name of the process. The keyword SELF refers to the contents of this location. The base of the data segment is defined by the value of r0. When the operating system kernel is

called to create a new process or to transmit initial parameter values to a created process, the value of r1 is a displacement in the data segment locating the parameters describing the operation. The process assumes that the operating system kernel will return any results starting at this displacement. When the operating system kernel returns control to the process, the process assumes that the success or failure of the operation is signalled by the value of r1. A value of -1 (0177777) indicates success; a value of 0 indicates failure.

Figure 2-3 illustrates the process data segments before and after the operating system kernel has been requested to create a new process.

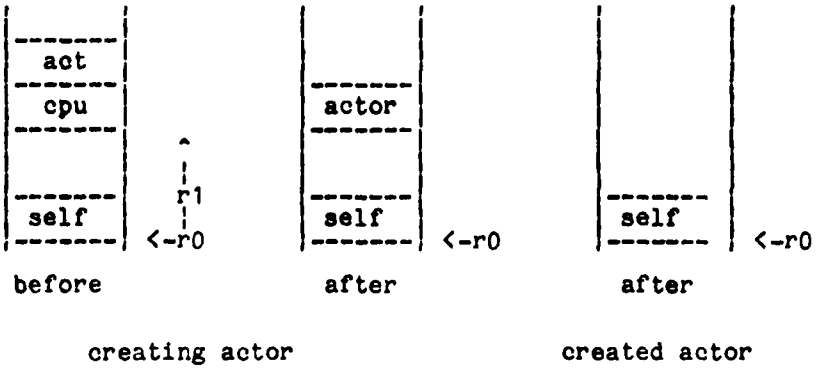


Figure 2-3
Data Segment Contents When Creating a New Process

The parameter "cpu" specifies the index of the CPU assigned to execute the new process. A negative value indicates that the EPL program has not specified an index. The parameter "act" specifies the address of the first executable instruction of the code defining the newly created process. The general structure of a code segment is the same as that of previous implementations and is discussed in [Fontaine 80]. The parame-

ter "actor" is a result returned by the operating system kernel and specifies the name of the newly created process.

Figure 2-4 illustrates the process data segments before and after the operating system kernel has been requested to transmit initial parameter values to a newly created process. The parameter "dst" speci-

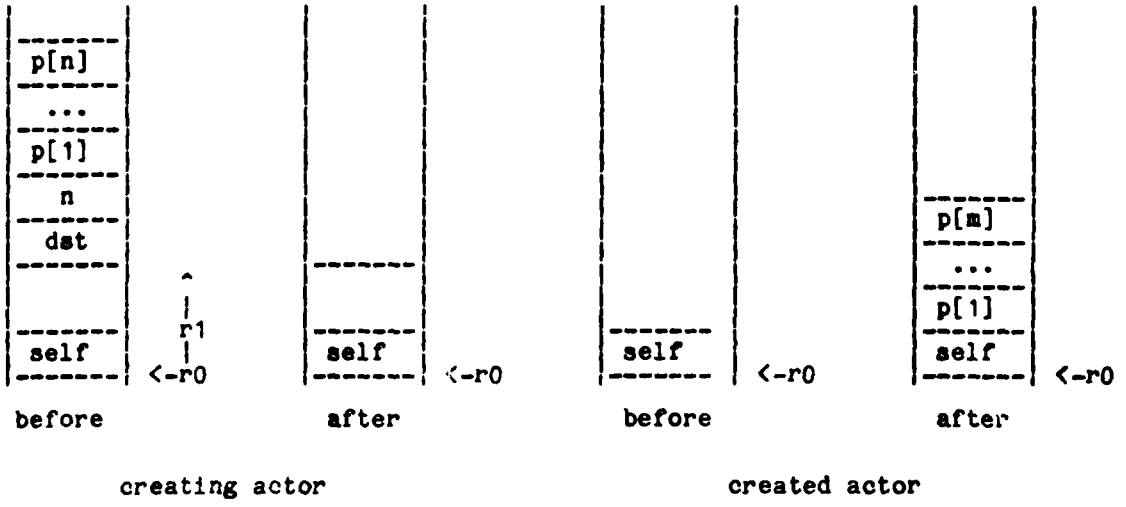


Figure 2-3
Data Segment Transmitting Initial Parameters to a Newly Created Process

fies the name of the new process waiting to receive initial parameters. The parameter "n" specifies the number of parameters to be transmitted. The parameters p[1] ... p[n] are the values to be transmitted. The parameters p[1] ... p[m] are the values that are received to initialize the new process. The value of "m" is that specified in the code segment of the process [Fontaine 80].

2.1.3 Example

Hoare [Hoare79] shows that processes can be used to encapsulate elements of a data structure so that the collection of processes representing a single instance of a data abstractions (such as the small set of integers) can be logically distributed among the elements of a distributed computer system. In EPL, a process is created for each element of the information structure. Logical distribution of the resulting software structure suggests that adjacent elements of the data structure be resident at separate processing sites. The code fragment shown in figure 2-4 is intended to suggest how EPL language features can be used to achieve this goal. In this example, an attempt is made to

---

```
LET [this.cpu, max.cpu] = [...]
 ...
n := this.cpu
REP
  n := (n REM max.cpu) + 1
  ACTOR a CPU n : [...] :: act.name ONFAILURE LOOP
  BREAK []
PER
 ...
```

Figure 2-4

Example of Process Creation Features

---

create the new process starting at the next available processing site. Should the attempt to create the process fail, the algorithm continue trying to create the new process at a site other than its own. The site of the parent process is selected only as a last resort and is assumed

to always be able to create a new process.

2.2 INTERPROCESS COMMUNICATION

The processes of EPL programs interact by transmitting and receiving messages. The message transmission primitives are synchronous, in that both the source and destination of a message must issues commands to send and receive data before the communication takes place. Thus, there is no requirement for the operating system kernel to buffer messages. This same mechanism also serves as the only synchronization device available to processes.

There are five commands that implement the transmission of data between processes: REC, RECF, REPLY, SEND, and procedure call. The current BNF specifies these message commands as follows

REC { <identifier> : } <idlist> THEN <tertiary>

RECF <secondary> : <idlist> THEN <tertiary>

REPLY <tertiary>

SEND <command> : <tertiary>

<identifier> [<parameters>]

The REC command specifies that the process will wait to receive a message from any source. The name of the source is optionally returned in the identifier. The scope of the identifier and the idlist is the tertiary following the keyword THEN. The RECF command indicates that the process will wait to receive a message from a specific source. The scope of variables is identical to that of the REC command. The REPLY command specifies that the process will wait to send a reply to a process. The destination of the message is the source of the message received in the smallest textually enclosing REC command. The SEND command indicates that the process will wait to send a message to a specific destination. The procedure call is a language concept for a commonly occuring programming construction used to implement procedure calls. It

has the effect of first SENDing a message to the destination named by the identifier and them immediately RECFing a reply from that process.

A common extension was made to each of these language concepts dealing with interprocess communication. This change is the option to specify a response if the operating system kernel detects a failure in sending or receiving messages. The detection of such failures is discussed in a later section.

2.2.1 Syntax and Semantics

The key word ONFAILURE as also been used in this context to delimit commands that define the response to failure. Each of the statements dealing with interprocess communication was modified to include an option specifying such response. The syntax of the language has been extended as follows

REC { <identifier> : } <idlist> THEN <tertiary> { ONFAILURE <tertiary> }

RECF <secondary> : <idlist> THEN <tertiary> { ONFAILURE <tertiary> }

REPLY <tertiary> { ONFAILURE <tertiary> }

SEND <command> : <tertiary> { ONFAILURE <tertiary> }

<identifier> [<parameters>] { ONFAILURE <tertiary> }

Any failure to transmit and/or receive values from another process will cause the tertiary following the ONFAILURE statement to be executed immediately. If an ONFAILURE tertiary is not specified and a failure is detected, the default action is to terminate the transmitting or receiving process. The scope of variables has been extended to include the tertiary following the key word ONFAILURE. The semantics of the REC, RECF, REPLY, and SEND commands is shown in figure 2-5. In each of these cases, a detectable failure causes the tertiary following the ONFAILURE key word to be executed. If the ONFAILURE option is not specified and a

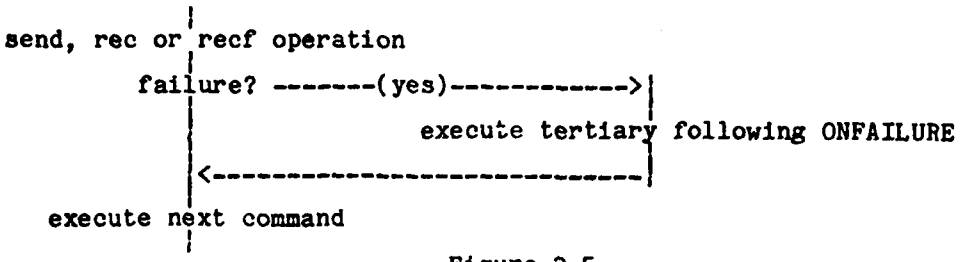


Figure 2-5

Semantics for the REC, RECF, REPLY, and SEND Commands

failure is detected, then the process is terminated. Figure 2-6 illustrates the semantics of the procedure call command. This command re-

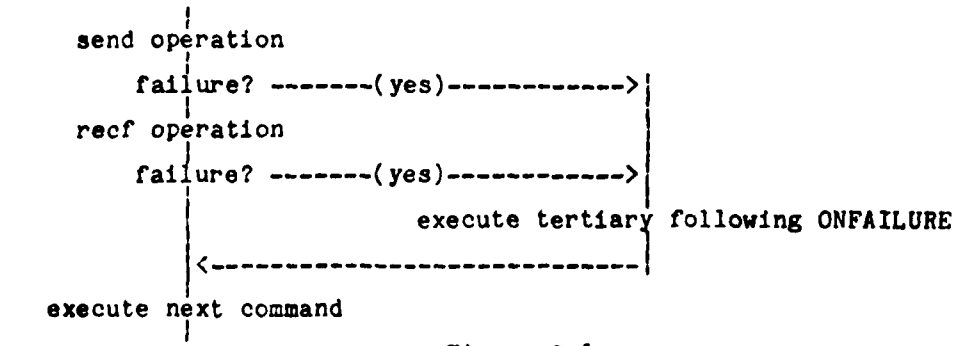


Figure 2-6

Semantics of the Procedure Call

quires the transmission and receipt of a message by the process executing the command. If either the transmission or the receipt of a message detects a failure, then the ONFAILURE tertiary is executed. No attempt

is made to distinguish between failures occurring during transmission and reception. If the ONFAILURE tertiary is not specified, and either component of the command fails, then the process is terminated.

2.2.2 Implementation

The LSI implementation of these concepts continues to use the following operating system kernel function to implement the transmission of messages due to a SEND, REPLY, or procedure call

ssend: send a message to a specified process

The REC and RECF commands use a modified forms of an earlier operating system function. These EPL commands are translated into distinct kernel functions. This was done to relax a previous compiler limitation on the names that could be used to represent processes. Specifically, the name 0 can not be used in previous implementations of the language. The REC and RECF are now respectively implemented by the following by the following operating system kernel functions

srec: receive a message from any source

srecf: receive a message from a specified source

The LSI-11 implementation of these functions has modified the format and interpretation of parameters that are passed between a process and the operating system kernel. When a process wishes to send or receive data, the value of r1 is used as a displacement in the data segment that identifies the location of parameters describing the operation. The process assumes that the operating system kernel will return any results starting at this displacement. When the operating system kernel returns control to the process, the process assumes that the success or failure of the operation is signalled by the value of r1. A value of -1 (0177777) indicates success; a value of 0 indicates failure.

Figure 2-7 illustrates the process data segments before and after the operating system has been requested to transmit or receive a mes-

sage. The parameter "dst" specifies the destination of the message to

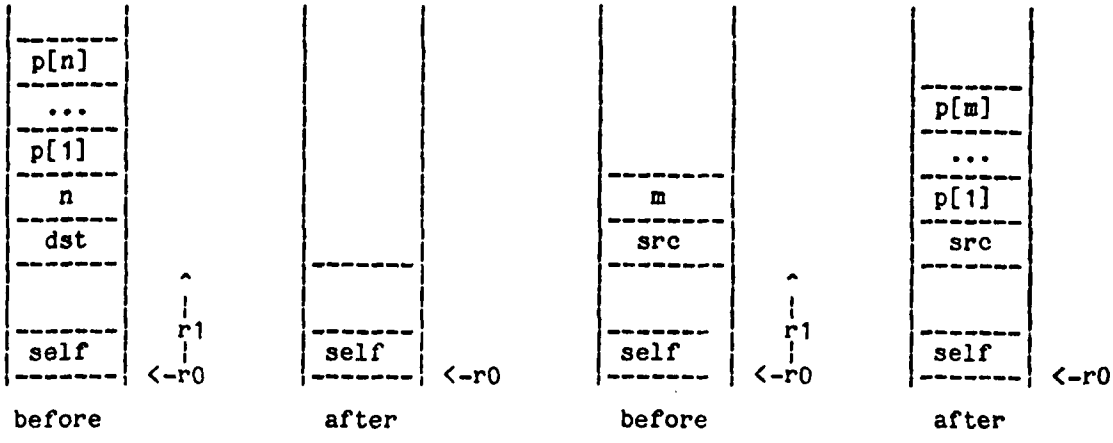


Figure 2-7
Data Segment for Processes Transmitting and Receiving Messages

be transmitted. The parameter "n" specifies the number of parameters to be transmitted to the destination process. The parameters p[1] ... p[n] are the values to be transmitted to the destination process. The paramter "src" specifies the source of the message in the case of the srecf function of the operating system kernel. Its value is not interpreted by the srec function of the operating system kernel. When the kernel functions of srec and srecf complete, the value of "src" the name of the process that sent the message. The parameter "m" specifies the number of parameters that the process expects to receive. The parameters p[1] ... p[m] are the values received by the destination process.

2.2.3 Example

Consider a software structure in which several identical processes are available are available to perform a particular function. If these processes are resident at separate processing sites, then the function provided by these processes has been replicated. Should one of the processes become inaccessable, then another process can be selected to perform the function.

Figure 2-8 illustrates how the EPL language extensions can be used to express such a software structure. The initial portion of the code shows the creation of multiple instances of a function. There is one instance per processing site. The process names are assigned to elements of a vector f. The latter portion of the code shows how the function can be used. A default (0) instance of the function is referenced. Should it be inaccessable, then the remaining instances of the function are accessed until an instance successfully provides results. Should all instances of the function become inaccessable, then the calling process terminated.

This example is trivial, in the sense that the function is assumed to produce outputs that are only a function of its inputs. A more important case occurs when the function is assumed to produce outputs that depend on its internal state. Replacating state-dependent functionality requires significantly more complicated communictions between processes than those used to implement a function call. The principles for designing such algorithms are themselves the subject of future research.

3.0 FAULT DETECTION

The language extensions presented in section 2 define a virtual machine that provides a specific format for reporting detectable errors to an application program (written as an EPL program). This format allows any process of an application program to determine if an interaction with another process succeeds or fails. This approach assumes that algorithms implemented as hardware or software (of the operating system kernel) can detect and isolate low-level errors, and that these errors

```
LET f = VEC size
LET i = 0
 ...
// initialize the vector f to the names of all processes capable
// of providing the same software function
i := 0
REP
  f!i := (ACTOR a CPU i : [...] :: act.name)
  i := i + 1
  IF i = size THEN BREAK [] FI
PER
 ...
// example of function use
i := 0
REP
  output.parameters := f!i [input.parameters] ONFAILURE (
    i := i + 1
    IF i = size THEN FINISH
    ELSE LOOP FI
    )
  BREAK []
PER
 ...
```

Figure 2-8

Replicated Functions

can always be mapped into the failure of a virtual machine instruction

governing process interactions.

This section of the report will present detectable errors that can be mapped into failures of the virtual machine instructions discussed in the previous section. Several further assumptions are made inorder to limit the scope of this discussion to failures that are clearly the responsiblity of the operating system kernel functions implementing these virtual machine operations. These assumptions are:

(1) the hardware of the distributed computer system is based on a direct interconnection structure [Anderson75] whose implementation is decentralized (e.g., Ethernet [Metcalf76]),

(2) detectable hardware errors render an entire processing site unusable,

(3) faults detected within a single process (e.g. addressing error) are either recovered by the process or cause the process to terminate.

The first of these assumptions eliminates the need to consider elaborate routing algorithms designed to recover from failures of paths in a communication network. The second assumption is overly strong. It is designed to simplify and focus the error detection algorithms of a single computing site to those required to support the interaction of computing sites in a distributed system. The third assumption makes a process responsible for only its own behavior and provides a uniform way of recovering from faults that directly impact only one process.

Given these assumptions, the following section (3.1) discusses how process interactions can fail. Given these modes of failure, section 3.2 discusses some simple approaches to implementing failure detection and reporting.

3.1 FAILURE MODES

The failure modes of process interactions can be systematically ex-

amined by considering each of the virtual machine operations defined by EPL. The analysis of failure modes also assumes a kernel-level protocol [Balkovich 80] that is symmetric between sending and receiving processes. I.e., either the source or destination may initiate a communication and enter a waiting state until the other party is ready to complete the communication.

This study approaches response to failure at the system level rather than at the level of individual processing sites or processes. The unit of hardware failure is a complete computing site, and the unit of software failure is a complete process. Loss of a processing site results in the loss of zero or more active processes.

3.1.1 Interprocess Communication

Communication between processes is defined by three virtual-machine operations: SSEND, SRECF, and SREC. These operations allow a process to transmit a message to a specific process, and to receive a message from a specified or unspecified process. Each of these operations may fail in one of the following ways:

(1) the source or destination of the message is invalid. This may result from a programmer error (forging a process name) or from data that becomes corrupted when transmitted between processes (yielding an undeefined process name),

(2) the source or destination process of the message has terminated,

(3) the site of the source or destination process fails before communication is initiated, or

(4) the site of the source or destination process fails after the process initiating the communication enters a waiting state.

3.1.2 Process Creation

Creation of a new process is defined by two operations: SCREATE and SRUN. These virtual machine operations allow an application to allocate space for a new process and to transmit initial parameter values to that new process. These operations may fail in the following ways:

(1) the site of creation or the destination process for parameters is invalid,

(2) the site of creation does not have sufficient resources to create the specified process,

(3) The site of creation fails before the creation was requested, or

(4) the site of the new process fails after creation of the new process, but before parameters are transmitted.

3.1.3 Issues

The problem of invalid process names can be approached in several ways. The simplest method is to validate that a named process has been created and that it has not yet terminated. A more comprehensive solution would be to treat process as capabilities [Denning76]. Using this approach one can validate both the names of processes and the virtual machine operations involving them. The use of capabilities was not explored as part of this work since the approach would require significant revisions to a language like EPL to restrict process interactions so that capability lists could be constructed and maintained.

Although there are several ways in which an interaction of processes can fail, the virtual machine supporting EPL is not used to distinguish between them. For example, any failure during the "simultaneous" creation of processes results in a failure to create all processes. This decision to not descriminate failure modes was based on the view that processes should be autonomous objects with little under-

standing of the internal state of other processes. If the need to distinguish the cause of failures becomes important, alternative language constructions are possible using the virtual machine operations that have been defined. At present, these new programming concepts of EPL, and constructions that can be generated using them, form a starting point for further evaluation of the programming concepts needed to decentralize software.

3.2 DETECTION METHODS AND RESPONSE TO FAILURES

Each failure mode can be detected using several approaches. Furthermore, each detectable error can be responded to in more than one way at the operating system kernel level. This section discusses the alternatives for each of the failure modes that have been identified.

3.2.1 Invalid Site or Process Identification

An invalid site specification is possible with all operations involving process creation or interprocess communication. As noted earlier, this may be the result of a programmer error or data that was corrupted during transmission between sites and subsequently used to identify a process.

An invalid site identification can be detected by the link-level communications software of the operating system kernel. Site addresses can be screened prior to transmission of kernel-level messages. An invalid site address should be reflected to the kernel-level software as a failed atttempt to transmit a kernel-level message.

An invalid process identification can only be detected by the operating system kernel receiving a kernel-level message. Screening for valid process identifiers can be centrally performed by the link-level software supervising the receipt of kernel-level messages. A kernel or link-level protocol involving acknowledgements is implied if such failures are to be made known to the site transmitting the message.

The response to an invalid site or process identifier is the responsiblity of the site transmitting the kernel-level message. The alternative responses to such failures are: (1) causing the specified virtual machine operation to fail at the application-level, and (2) causing the process that supplied (to the kernel) the invalid parameter to involuntarly terminate. The latter response assumes that the invalid parameter is a symptomatic of erronous process behavior and forcibly terminates that behavior. The former view assumes that the application programmer may wish to specify an algorithm for recovering from such failures (the default algorithm is to terminate the process). The first approach is more flexible and therefore perferable in an experimental system.

3.2.2 Communication with a Terminated Process

It is possible a process that has terminated is the source or destination of a message in a SSEND or SRECF operation. This may be the result of a programmer error at the application-level. This may also be the result from failures such as those described in the preceeding section, when the response to failure is to terminate a process.

Process termination can occur at one of two times with respect to communication. A process may terminate prior to being named as the source or target of a communication with another process. A process may also terminate after being named as the source or target of a communication, but before that communication is complete. This case arises when a multiple-message protocol is used to implement kernel-level interactions. An example of such a protocol is given in [Fontaine 80]. In the latter case, the partner in the communication is in a blocked state.

A process terminating prior to being named as the source or target of a communication by another process can be detected by the operating system kernel supervising the terminated process. Detection occurs when the first kernel-level message arrives indicating that another process wishes to communicate with the terminated process. This assumes that

the kernel-level protocol implementing interprocess communications is symmetric for the receiving and sending sites. When a receiving process initiates the interaction (srecf), it must be able to interogate the sender's site to determine if the sending process has terminated. This requires a more complex protocol than that described in [Fontaine 80]. Since detection of this fault depends on interogation of process state information, detection should take place in the kernel-level routines dealing with the sending and receiving of application-level messages.

A process that terminates after being named as the source or target of an incomplete communication requires a more complicated response. Process(es) blocked, waiting to communicate with the terminating process, must be notified of the termination. Since detection of this fault depends on interogation of process state information, detection should take place in the kernel-level routines dealing with the termination of a process. Detection requires that a message be broadcast to all sites with processes (potentially) waiting to communicate with the terminated process, notifying them of the process termination. If a process terminates before or during an interaction with another process, it should result in a failure of the corresponding virtual machine instructions (i.e., ssend, srecf). This allows an application-level program to respond to failures by rerouting application-level interactions.

3.2.3 Site Failures

It is possible to specify a site that has failed as the source or destination of any kernel-level message. A site may fail at one of two times with respect to virtual machine operations implemented by the operating system kernel. The site may fail prior to being named as the target of the kernel-level message initiating an interaction. The site may also fail after the process initiating the interaction has been placed in a blocked state pending a response of a process at the failed site (e.g., SSEND, SRUN, SCREAT, and SRECF operations).

A site that fails prior to being named as the target of a kernel-

level message can be detected by the operating system kernel supervising the process initiating the communication. Detection can be accomplished by a link-level protocol when the first kernel-level message is transmitted to the failed site. This requires a link-level protocol based on acknowledgements and time-outs.

A site that fails after processes at other sites are in states waiting for responses is more complicated. This type of error must be detected by the sites supervising the blocked processes. Two approaches are possible. A blocked process can be allowed to remain in a blocked state for a fixed period of time. Failure to obtain a response in a fixed amount of time is assumed to indicate that the other site associated with the operation has failed. Alternatively, adjacent sites can be periodically interogated to determine if they have failed. The first approach assumes that a suitable time-out interval can be defined for all waiting processes. Since this depends on the nature of the application, it must be chosen large enough to assure that site failures will not be detected when the application requires an extensive amount of time before responding to a request. This approach is undesirable because it provides very poor response to failures and because it requires a significant overhead to administer timers associated with waiting processes. The second approach can provide a more prompt response to failures and does not require administering timers for each waiting process. However, it does introduce an overhead associated with the periodic interogation of sites.

The response to a site failure is the responsibility of every other site of the system. The only reasonable response to a such a failure is to cause all subsequent virtual machine operations and any blocked virtual machine operations associated with that site to fail. This allows the application programmer to specify an algorithm for recovering from such failures (the default algorithm is to terminate the process requesting the operation).

3.2.4 Insufficient Resources

A request to create a process may be find that there are insufficient resources to create the process at a particular site. In the design described in previous sections, this may be the result of exhausing process identifiers or exhausting the memory available for process data segments.

Insufficient resources can be detected by the operating system kernel at the target site of the creation. Since detection requires access to local state information, detection should be the responsibility of kernel-level routines. This type of failure should be reflected to the operating system kernel requesting the creation by extending the kernel-level protocol to include negative acknowledgements of the creatation request.

The response to insufficient resources is the responsibility of the operating system kernel administering the parent process. The appropriate response is to cause the virtual machine operation to fail. This assumes that the application programmer will specify an algorithm for recovering from such failures (the default algorithm is to terminate the parent process).

4.0 BIBLIOGRAPHY

[Anderson75] Anderson, G.A., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", ACM Computing Surveys, (7,4), December 1975, pp. 197-213.

[Chu80] Chu, W.W., et.al., "Task Allocation in Distributed Computing", Computer, (13,11), November 1980, pp. 57-69.

[Denning76] Denning, P.J., "Fault-Tolerant Operating Systems", ACM Computing Surveys, (8,4), December 1976, pp.359-389.

[Fontaine80] Fontaine, S., An Operating System Kernel for a Distribute Computer System, Technical Report Department of Electrical Engineering

and Computer Science, University of Connecticut, Storrs, CT, (in preparation).

[Hoare79] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, (21,8), August 1978, pp. 666-677.

[Jones80] Jones, A.K., and K. Schwans, "Experience Using Multiprocessor Systems: A Status Report", ACM Computing Surveys, (11,2), June 1980, pp. 121-165.

[LeLann74] LeLann, G., "Distributed Systems - Towards a Formal Approach" Proceedings of the IFIP 74 Congress, North Holland, 1977, pp. 155-160.

[May78] May, M.D., R.J.B. Taylor, and C. Whitby-Strevens, "EPL - An Experimental Programming Language", in IEEE Conference on Trends and Applications: Distributed Processing, Gaithersburg, MD, May 1978, pp. 68-71.

[May79] May, M.D., and R.J.B. Taylor, The EPL Programming Manual, Report No. 7, Department of Computer Science, University of Warwick, Coventry, England, 1979.

[Metcalf76] Metcalf, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, (19,7), July 1976, pp. 395-404.

[Taylor80] Taylor, R.J.B., and J.R.N. Lowe, Notes on Programming in EPL, Department of Computer Science, University of Warwick, Coventry, England, (in preparation).

4.0 BIBLIOGRAPHY

[Anderson75] Anderson, G.A., "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples", ACM Computing Surveys, (7,4), December 1975, pp. 197-213.

[Chu80] Chu, W.W., et.al., "Task Allocation in Distributed Computing", Computer, (13,11), November 1980, pp. 57-69.

[Denning76] Denning, P.J., "Fault-Tolerant Operating Systems", ACM Computing Surveys, (8,4), December 1976, pp.359-389.

[Fontaine80] Fontaine, S., An Operating System Kernel for a Distribute Computer System, Technical Report Department of Electrical Engineering and Computer Science, University of Connecticut, Storrs, CT, (in preparation).

[Hoare79] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, (21,8), August 1978, pp. 666-677.

[Jones80] Jones, A.K., and K. Schwans, "Experience Using Multiprocessor Systems: A Status Report", ACM Computing Surveys, (11,2), June 1980, pp. 121-165.

[LeLann74] LeLann, G., "Distributed Systems - Towards a Formal Approach" Proceedings of the IFIP 74 Congress, North Holland, 1977, pp. 155-160.

[May78] May, M.D., R.J.B. Taylor, and C. Whitby-Strevens, "EPL - An Experimental Programming Language", in IEEE Conference on Trends and Applications: Distributed Processing, Gaithersburg, MD, May 1978, pp. 68-71.

[May79] May, M.D., and R.J.B. Taylor, The EPL Programming Manual, Report No. 7, Department of Computer Science, University of Warwick, Coventry, England, 1979.

[Metcalf76] Metcalf, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, (19,7), July 1976, pp. 395-404.

[Taylor80] Taylor, R.J.B., and J.R.N. Lowe, Notes on Programming in EPL, Department of Computer Science, University of Warwick, Coventry, England, (in preparation).

Appendix A

DISTRIBUTED COMPUTING PROJECT REPORT #1

The EPL Programming Manual

by

M.D. May and R.J.B. Taylor

Department of Computer Science
University of Warwick
Coventry CV4 7AL

May 1979

INTRODUCTION

EPL is an experimental progamming language designed to aid research into distributed computer systems.

Some of the distinguishing features of EPL are:

An EPL program is expressed as a number of Acts, which are performed by Actors.

Actors communicate with each other only by sending messages.

Acts may be nested and Actors can be created by other Actors providing both hierarchy and parallelism.

The basic data object is a word with no particular disposition as to type. For example, a word may be regarded as a bit-pattern, a number, or the name of an Actor. No attempt is made to enforce type restrictions either at compile-time or at run-time. In this respect EPL has both the flexibility and pitfalls of machine language.

This manual is not intended as a primer; the constructs of the language are presented with scant motivation and few examples. To the experienced and disciplined programmer it is a powerful and useful language but there are few provisions for the protection of naive users.

The language described here is based on that described in "The EPL Programming Manual" by M.D.May (Sept 1976), which in turn was based on BCPL. The current version of EPL differs from the original in that several features of the original have not been implemented and the syntax has been simplified.

Language Definition

2.1 Notation

The syntax of the language is given in an extended Backus-Naur form (BNF). Nonterminals are enclosed in the sharp brackets '<' and '>'. A production rule consists of:

<nonterminal> ::= <production>

.ctions may contain terminals and nonterminals. A number of symbols with special meanings are used in productions. The symbol '|' denotes alternatives. Symbols enclosed in curly brackets '{' and '}' are optional. A star after the closing curly bracket '}*' denotes that the symbols within the brackets may be repeated any number of times, or omitted.

2.2 Elements

EPL is a language which manipulates values. A <value> is any value which can be represented as a bit pattern in one computer word. In various contexts values are interpreted in different ways, for instance as integers, characters or pointers to objects.

A value can be explicitly represented in a number of ways, or associated with an identifier, and these form the elements on which the language operates.

```
<element> ::= <identifier> | <value>
<value>   ::= <number> | <string constant> | <character constant>
            | TRUE | FALSE
```

An <identifier> consists of a sequence of letters, digits and dots, the first character of which must be a letter.

A <number> can be represented in several bases. A decimal number consists of a sequence of decimal digits. All numbers in other bases consist of a '#' symbol, followed optionally by a character to denote the base, followed by a sequence of digits in that base.

Bases are signified by the following characters:

Letter	Base	Allowed Digits
B	Binary	0 1
O	Octal	0 1 2 3 4 5 6 7
X	Hexidecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

If the letter signifying the base is left out, the base defaults to Octal.

The value FALSE is represented as a number and the representation of TRUE is the logical complement of FALSE.

A <string constant> consists of up to 255 characters enclosed in string quotes ("). The character " may be represented only by the pair *" and the character * can only be represented by the pair **.

Other characters may be represented as follows:

*N	is newline	*C	carriage return
*T	is horizontal tab	*G	bell
*S	is space		
*B	is backspace		
*P	is newpage		

The machine representation of a string is the address of the region of store where the length and characters of the string are packed.

A <character constant> consists of a single character enclosed in character quotes ('). The character ' can be represented in a character constant only by the pair *'. Other escape conventions are the same as for a string constant. A character constant is right justified in a word.

2.3 Expressions

All constructs of EPL are defined to return results, so the general syntax and semantics are described here.

Depending on context a result may be a single value or a multiple value. Although each construct returns a result, the use to which the result is put depends upon the environment of the construct. In many cases the result is discarded.

The BNF for expressions is as follows:

```
<base>          ::=  <element> | <clause> | <procedure call>
<primary>       ::=  { <monadic op> }* <base>
<secondary>     ::=  <primary> { <diadic op> <primary> }* | <vat>
<constant>      ::=  <secondary>
<command>       ::=  <message command>
                   |  BREAK <tertiary>
                   |  SYS <tertiary>
                   |  LOOP
                   |  <lhs> := <tertiary>
                   |  <secondary>
<tertiary>      ::=  <command>
                   |  [ { <commmand> { , <command> }* } ]
                   |  <declaration>
<lhs>           ::=  <lhs element>
                   |  [ <lhs element> { , <lhs element> }* ]
<lhs element>   ::=  <identifier> | <identifier> ! <secondary>
<serial>        ::=  <tertiary> { ; <tertiary> }*
<clause>        ::=  ( <serial> )
                   |  REP <serial> PER
                   |  IF <serial> THEN <serial>
                                  { ELSE <serial> } FI
                   |  CASE <serial> IN <case serial>
                                  {  OUT <serial> } ESAC
<case serial>   ::=  <numbered pack> { , <numbered pack> }*
<numbered pack> ::=  <constant> : { <constant> : }* <tertiary>
```

The BNF and explanation for <vat>, <declaration>, <message command> and <procedure call> are given in the appropriate sections.

2.3.1 Primaries

A primary is a base optionally preceded by monadic operators. The monadic operators are

```
+       plus
-       minus (arithmeticcomplement)
\       not (l-   al complement)
```

The result returned by a primary preceded by an operator is a single value.

2.3.2 Secondaries

A secondary is a primary optionally followed by a sequence of diadic operators and primaries. A secondary is evaluated from left to right except where

the precedence of the operators dictates otherwise. The following are the diadic operators given in order of precedence, highest (most binding) first.

Operators	Comment
!	subscription
<<, >>	left shift, right shift
/\	and
\/	or
*, /, REM	multiply, divide, remainder
+, -, EQV, NEQV	plus, minus, equivalence, not equivalence
<, <=, >, >=	less than, less than or equal, greater than, greater than or equal
=, /=	equal, not equal
ANDF	and if
ORF	or if

The arithmetic operators (+, -, *, /, REM) operate on single values and yield a single value; overflow is undefined. Divide (/) operates on integers and yields an integer dividend, while REM yields the integer remainder after division. Divide and REM operate so that the following equation is always true whatever the values of a and b.

```
a = (a / b) * b + a REM b
```

If both operands are positive the value yielded by REM is positive.

The relational operators (<, <=, >, >=, =,\=) compare their two operands and yield either TRUE or FALSE. Inequalities (<, <=, >, >=) should not be used to compare values other than integers and characters.

The logical operators (/\,\/, EQV, NEQV) operate on the whole of their operands, and on each bit independently to produce a result bit.

The shift operators (<<, >>) yield the left hand operand shifted to the left or right by the number of places given by the right hand operand. Vacant bits are filled with zero.

The subscription operator (!) takes the left hand operand as a pointer to a VEC or TAB (see 2.4.2) and subscripts it by the right hand operand.

The truth value operators (ANDF, ORF) work in the following ways. For ANDF the left hand operand is evaluated and if it is FALSE the value yielded is FALSE, otherwise the value yielded is the right hand operand. For ORF the left hand operand is evaluated and if it is TRUE the value yielded is TRUE, otherwise

the value yielded is the right hand operand. The truth value operators are so called because they can yield a truth value without the whole of an expression being evaluated.

The result returned by a secondary which contains diadic operators is a single value.

2.3.3 Constants

A constant is a <secondary> which contains only numbers, character constants, names declared in previous constant declarations, TRUE, FALSE and the operators *, /, REM, + and -.

2.3.4 Tertiaries

A tertiary yields a result which may be

- Empty (written as [ ]).
- A single value
- A multiple value

The result is normally enclosed in square brackets ([ ]), but if it is a single value they may be omitted.

A multiple value is either an expression whose result is a multiple value, or a set of expressions which yield single values enclosed in square brackets. If a multiple value is yielded in a context which requires a single value (for instance in a secondary which contains diadic operators or as one of the expressions within a tertiary) the result is the first value of the multiple value.

The order in which the values of a multiple value are evaluated is not defined.

2.3.5 Clauses

Clauses are constructed from serials. The declarations and tertiaries of a serial are evaluated in sequence, and the result of a serial is the result of its final tertiary. There are four different types of clause, the simplest is a serial enclosed in parentheses.

REP PER delimit a serial that is continually repeated until a BREAK is en-

countered within it (see 2.3.6).

The IF and CASE clauses both select an expression to be evaluated. In an IF clause the serial following the IF is evaluated. If its value is TRUE the result of the IF clause is the result of the serial following THEN, otherwise it is the result of the serial following ELSE, or, if there is no ELSE, the result is not defined.

In a CASE clause the serial following the CASE is evaluated to give a number which is matched against constants in the numbered packs following the IN. If a match is found, the result of the CASE clause is the result of the corresponding tertiary in the numbered pack. If no match is found the result of the CASE clause is the result of the tertiary following OUT, or, if there is no OUT, the result is not defined.

2.3.6 Commands

Description of <message command> is left until Acts and Actors are introduced in section 2.5.

BREAK and LOOP are both commands used only in REP PER loops. BREAK <tertiary> forces an exit from the loop yielding the result of the tertiary. LOOP causes execution to resume after the REP and is used to repeat the loop. Both operate only on the smallest textually enclosing loop.

Assignment (:=) allows single and multiple values to be assigned to variables. Assignments are simultaneous. This means that the entire right hand side is evaluated and then assigned to the left hand side, the first element of the multiple value to the first element on the left hand side, the second to the second , and so on. The left hand side may only consist of variables and subscripted variables.

There are no requirement that the left hand side and the right hand side have the same number of elements in a multiple assignment. However, if the right hand side has more elements than the left hand side the surplus values are ignored in the assignment. If the left hand side has more elements than the right hand side, the values assigned to the surplus variables are undefined. The result yielded by an assignment is the result of the right hand side.

SYS causes the tertiary to be passed as parameters to a procedure in the runtime system. The effect of this command is implementation dependent.

2.4 Declarations

There are two kinds of declaration in EPL. The CONST declaration introduces constants. The LET declaration, together with the ACTOR declaration, the parameters of an ACT and the Receive command (which are described in section 2.5), introduces variables.

In EPL the scope of a declaration is the <declaration> in which it appears, and any subsequent <declaration> and <tertiary> (including any constructs within them) in the <serial> in which it appears. Acts further restrict the scope of variable declarations (see section 2.5.1). If an identifier is used again in a declaration in an inner serial, the scope of the original declaration does not extend to any construct in that serial after the <declaration> in which the identifier is reused. An identifier cannot be declared more than once in a serial.

The result returned by a declaration is undefined.

The BNF for declarations is as follows:

```
<declaration>   ::=  CONST <const decl> { , <const decl> }*
                   ¦ LET <idlist> = <decl pack>
                   ¦ <actor declaration>
<const decl>    ::=  <identifier> = <constant>
                   ¦ <identifier> = <act declaration>
<idlist>        ::=  <identifier>
                   ¦ [ <identifier> { , <identifier> }*]
<decl pack>     ::=  <secondary>
                   ¦ [ <secondary> { , <secondary> }* ]
<vat>           ::=  VEC <constant>
                   ¦ TAB <tertiary>
                   ¦ <act declaration>
```

The decriptions of <actor declaration> and <act declaration> are given in section 2.5.

2.4.1 CONST Declarations

A constant declaration consists of a list associating identifiers and constants. Within the scope of the declaration, an identifier may be used only in contexts where the associated constant would be allowable.

It is also possible to associate an Act with an identifier in a *CONST* declaration.

No result is returned.

2.4.2 LET Declarations

The LET declaration declares the names in the <idlist> as variables. Each variable contains one value and it is initialized from the <decl pack>. The order in which the variables are initialized is undefined.

There are further restrictions on the scope of a variable. See section 2.5.1.

The result is the value of the <decl pack>.

2.4.3 VEC and TAB

VEC and TAB both reserve contiguous areas of storage which can be accessed using the subscription operator. Each element of a VEC or TAB is large enough to hold one value. Noth constructs return a pointer to the area of reserved storage.

VEC reserves storage in such a way that the first element corresponds to subscript zero, and the subscript of the last element is given by the <constant> following VEC.

TAB reserves enough storage to hold the values of the <tertiary> following it. It is initialized to the values of the tertiary, and the subscript of the first element is zero. Each value in the tertiary must be a <constant>.

The effect of assignments to the elements of a TAB is undefined.

The result of a VEC or TAB is undefined.

2.5 Acts and Actors

Programs in EPL are expressed in terms of Actors, which communicate by sending messages to one another. Actors are created to perform Acts, and there may be any number of Actors performing the same Act. Actors start to perform their Act as soon as they are created, and die on completion of their Act.

The BNF for Actors, Acts and Messages is as follows.

```
<message command>     ::=  REC <message> THEN <tertiary>
                         |  RECF <message> THEN <tertiary>
                         |  SEND <message>
                         |  REPLY <message>
<message>             ::=  <secondary> : <tertiary>
<actor declaration>   ::=  ACTOR <actor> { , <actor> }*
<actor>               ::=  <identifier> : <tertiary> : : <command>
<act declaration>     ::=  ACT <idlist> : <tertiary>
<procedure call>      ::=  <identifier> [ ( <parameters> ) ]
<parameters>          ::=  <command> { , <command> }*
```

The definition of <message> is a general form. In specific contexts it is more restricted as will be shown.

2.5.1 Acts

An Act is a description of a computation. It is declared as

ACT <idlist> : <tertiary>

and this yields a reference to code compiled to evaluate the tertiary. The code is implementation dependant. The names in the <idlist> are declared as variable names for the parameters of the act. These parameters are initialized for an Actor of the Act when the Actor is declared.

An Act further restricts the scope of variable names. An Act cannot reference variable names declared outside that Act.

2.5.2 Actors

Actors are invocations of Acts. They are declared in ACTOR declarations which have the form

ACTOR <identifier> : <tertiary> :: <command> { , <actor> }*

The <identifier> is a variable which is initialized to refer to the Actor. The <tertiary> is used to initialize the parameters of the Act. The <command> is the Act which the Actor will perform. It must yield a value which is an Act.

The scope of an Actor variable includes the whole of the Actor declaration in which it is declared, and in particular it can be used as a parameter of other Actors, The Actors of an Actor declaration are declared in parallel and are created simultaneously. This means that several Actors can be declared at once which know about each other and can send each other messages.

The result is the name of Actor.

2.5.3 Receive Commands

The REC and RECF commands are used to receive messages. They may appear anywhere within the body of an Act. When a Receive command is obeyed within an Actor, its execution is halted until a message arrives (if a message is already waiting the Actor can continue). The full BNF is:

REC { <identifier> : } <idlist> THEN <tertiary>

RECF <secondary> : <idlist> THEN <tertiary>

The identifiers in the <idlist> are declared as identifiers exactly as in a LET declaration, and initialized with the received message. Their scope is the <tertiary> following the THEN.

The REC command receives messages sent from any other Actor. The identity of the Actor which sent the message is assigned to the identifier preceding the colon if it is present. RECF receives a message only from a specified Actor. This is given by the <secondary> which precedes the colon.

The REC and RECF commands yield the value of the tertiary following the THEN.

2.5.4 Send and Reply commands

A message is a set of values which is passed from one Actor to another. Send always waits for a receive. Any value can be passed but some values (for instance VECs or ACTs) may have no significance at their destination. A special type of send is the REPLY command. It sends a message to the Actor which sent the last message to the smallest textually enclosing REC command. The full BNF is:

SEND <command> : <tertiary>
REPLY <tertiary>

The message is formed by evaluating the <tertiary>. It is sent to the Actor given by the command preceding the colon, which must yield a value which is an Actor. Because this Actor is knowwn in the REPLY command it is not explicitly stated.

The result of SEND and REPLY are undefined

2.5.5 Procedures

A procedure call is of the form

<identifier> [<parameters>]

It is equivalent to the program fragment :-

```
SEND <identifier> : [ <parameters> ];
RECF <identifier> : [ a, b, c, . . . ] THEN  a, b, c, . . .]
```

The procedure call is used to call Actors which have already been created and is akin to the use of a Class in a language which provides for data abstraction.

2.6 Program

At the outermost level an EPL program consists of a sequence of declarations and initializing commands. There may be restrictions on the class of commands that can be obeyed at the outermost level.

2.7 Miscellaneous Features

2.7.1 Comments and Spaces

The character pair /* denotes the beginning of a comment. All characters from (and including) /* up to and including the character pair */ will be ignored by the compiler.

Blank lines are also ignored.

Space and tab characters may be inserted freely except inside an element, a system reserved word (e.g. THEN), or inside an operator (e.g. :=) Space or tab characters are required to separate identifiers or system words from adjoining identifiers or system words.

2.7.2 Synonymous Symbols

For convience some operators have a number of different representations They are:-

/\	&
\/	|
\	~
\=	~=

Discussion of the EPL Dictionary

1. A dictionary is always produced and saved.

2. EPL always tries to read a dictionary.

3. Any CONST declaration at the outermost level is entered in the dictionary.

   a. Manifest Constants are stored as constants
   b. ACTS are assigned a unique global symbol number

4. The unique numbering of ACTS depends upon serial translation of the ACTS.

5. An undeclared symbol is assumed to be an externally defined, yet to be compiled ACT. A warning message is issued.

6. Assumption 5 may impose ordering on computations if manifest constants are scattered through program modules.

7. The main actor is the one loaded first. All acts are suffexed with the empty tertiary, so that if a CONST is executed it does nothing.

8. The dictionary uses file names to keep track of where symbols were defined.

9. The dictionary is produced in SYN4 and consumed in TRN6 as well as updated.

10. We should restrict compiler to producing load modules only since execution is dependent on order of loading.

Appendix B

VIRTUAL MACHINE DEFINITION FOR EPL

S is a pointer to the first free location on the stack.
A points to the base of the actor: (S = @(A!1) initially).
+> is an addition operator whose left operator is an address and whose right operand is an integer.

Load Operations:

```
LA  n           !S := A!n;        S := S+1
LN  n           !S := n;          S := S+1
LACT  Ln        !S := Ln;         S := S+1
LGACT  GLn      !S := GLn;        S := S+1
TRUE            !S := true;       S := S+1
FALSE           !S := false;      S := S+1

LSTR  n         !S := @<string>;S := S+1
   C1 ... Cn     <string> consists of following n characters

LTAB  n         !S := @<tab>;     S := S+1
   V1 ... Vn     <tab> consists of following n constants

LVEC  n         !S := A +> n;     S := S+1
```

Store Operations:

```
SA  n           S := S-1;         A!n := !S
SINDEX          S := S-3;         !(!(S+1) +> !(S+2) := !S
```

Arithmetic Operations:

```
INDEX           S := S-1;         !(S-1) := !(!(S-1) +> !S)
MULT            S := S-1;         !(S-1) := !(S-1) * !S
DIV             S := S-1;         !(S-1) := !(S-1) / !S
REM             S := S-1;         !(S-1) := !(S-1) rem !S
PLUS            S := S-1;         !(S-1) := !(S-1) + !S
MINUS           S := S-1;         !(S-1) := !(S-1) - !S
EQ              S := S-1;         !(S-1) := !(S-1) = !S
NE              S := S-1;         !(S-1) := !(S-1) / !S
GR              S := S-1;         !(S-1) := !(S-1) > !S
LS              S := S-1;         !(S-1) := !(S-1) < !S
GE              S := S-1;         !(S-1) := !(S-1) _ !S
LE              S := S-1;         !(S-1) := !(S-1) _ !S
LSHIFT          S := S-1;         !(S-1) := !(S-1) << !S
RSHIFT          S := S-1;         !(S-1) := !(S-1) >> !S
AND             S := S-1;         !(S-1) := !(S-1) / !S
OR              S := S-1;         !(S-1) := !(S-1) / !S
EQV             S := S-1;         !(S-1) := !(S-1) _ !S
NEQV            S := S-1;         !(S-1) := !(S-1) _ !S
NEG                               !(S-1) := -!(S-1)
NOT                               !(S-1) := ~!(S-1)
```

Miscellaneous Operations:

```
JUMP  Ln      goto Ln
JT  Ln        S := S-1;       if !S goto Ln
JF  Ln        S := S-1;       unless !S goto Ln
Ln:           Ln:  // labelling program

GLn:          GLn: // labelling program with externally
                      available label

SYS  Lm n     call sys in runtime system (kernel) with
              A!n onwards as parameters.
              On failure goto Lm.
              Any results will be found on the stack from A!n on.

CASE  Ln  k   S := S-1;       k cases follow; Ln is the OUT label
  C1:  L1     if !S = C1 goto L1
   .              .
   .              .
   .              .
  Ck:  Lk     if !S = Ck goto Lk
STORE         directive to optimising code generator
              to ensure that items held in temporary
              storage are placed on the stack

STACK  n      S := A+n
```

Message Operations:

```
REC  Lk n m   Receive a message on the stack A!(n+1) to A!m.
              A!n initially holds the identity of the actor
              from which the message will come. If A!n = 0
              then the message can come from any actor and
              the identity of the sending actor is put there.
              A!(n+1) initially holds the number of items to be received.
              On failure goto Lk.
              The operation implies a STORE.
              S := A+m+1

SEND  Lm n    Send items A!n onwards as a message. A!n, A!(n+1)
              contain the identity of the actor to which
              the message is sent and the number of items in the
              message proper.
              On failure goto Lm.
              S := A+n

CREATE  Lm n  S := A+n+1;  !(S-1) := the identity of the actor created
              from the act at A!(n+1) on CPU number A!n (if A!n = -1
              any CPU may be chosen). The newly created actor has its
              own identity as its A!1.
              On failure goto Lm.

RUN  Lm n     Exactly the same as SEND except that the destination is
              a newly created actor and the message is the
              parameterisation of that actor.

ACT  Ln  p    Declares an act at label n.  Its parameters are A!2 to A!p;
              S := A+p+1;     and an implied STORE of its parameters is
              performed.  Note that A!1 contains name of self.
```

ENDACT Ln	Marks the end of an act. Ln is a label for premature premature termination of the act
END	Marks the end of the module.

Part V

On the Performance of Decentralized Software

by

E. Balkovich and C. Whitby-Strevens

ABSTRACT

Distribution of computing to achieve goals such as enhanced reliability depend on the use of decentralized software. Decentralization typically replaces a sequential process by a system of small, concurrent processes that interact frequently. The implementation of processes and their interactions represents a cost incurred as a result of decentralization. Performance measurements are reported in this paper for decentralized software written in a programming language for distributed computer systems. These performance measurements confirm that low-cost implementations of concurrency are possible, but indicate that decentralized software makes heavy use of run-time functions managing concurrency. An initial model comparing the performance of a specific decentralized software structure to its centralized counterpart indicates that these implementation costs are generally offset by the performance improvements that are due to the parallelism inherent in the decentralized structure. The research facilities for continued study of decentralized software performance are discribed in the summary.

KEY WORDS AND INDEX TERMS

Distributed Computer Systems, Decentralized Software, Decentralized Control, Performance Measurement and Evaluation, Concurrent Software

1 INTRODUCTION

Distributed computer systems are frequently proposed as alternatives to conventional, single computer systems for applications that require attributes such as high reliability, or incremental system growth. Most of these system attributes can only be achieved if both software and hardware are decentralized. For example, hardware reliability can be improved by using multiple computers to decentralize the computational capacity of a system. Unless software for such a system is also decentralized, the loss of a specific computer may preclude executing a crucial software component, causing the system to fail.

Decentralization of software effects both control and data structures. Approaches involving partitioning, replication, or circulation of state information are usually applied to decentralized a control algorithm (e.g., [4,5,11]). Information structures are generally decentralized by representing the elements of the structure as individual processes (e.g., [9,10]). Generally, a sequential process of a centralized implementation has as its counterpart, a set of interacting concurrent processes in a decentralized implementation. The resulting software structures can use large numbers of small processes that interact frequently. One concern frequently voiced about decentralized software is the potentially high cost of implementing large numbers of small processes.

There is a body of research whose goal is to define programming languages for distributed computer systems [1,3,7,9,12]. These languages can be used to express decentralized algorithms that will hopefully have low-cost implementations. The language features most commonly proposed include: (1) the ability to apply concurrency at one or more levels of the software structure (i.e., nested levels of concurrency), (2) restricted forms of process interaction that generally avoid shared variables, and (3) nondeterministic control structures such as the guarded command [6]. The implementation of such language features implies the existance of an operating system kernel that pro-

vides functions to manage concurrency, interprocess communication, and nondeterministic control.

The primary purpose of this paper is to report on findings that illustrate how decentralized software utilizes the functions provided by such an operating system kernel. These findings confirm the feasibility of low-cost implementations of language concepts that are expressly designed for distributed computers. These results also show that decentralized software will make extensive use of these functions. This usage normally represents a cost incurred by decentralization that would not be present in an equivalent centralized approach. Thus, it can be thought of as a trade-off required to achieve attributes such as enhanced reliability or incremental growth.

The next section discusses the features of a specific programming language used to write decentralized software. The implementation of this language is outlined to establish the functions that are encompassed by its implementation costs. The third section reports on performance measurements that describe how the implementation of this language is used by decentralized software. A final section discusses the application of these performance measurements and describes a research facility that will be used to support future studies.

2 BACKGROUND

The results of this study were derived by observing decentralized software written in a specific programming language -- EPL. Section 2.1 provides an overview of the features of this language and an illustration of its use. Programs written in EPL require a run-time environment that manages process and their interactions. These functions are provided by an operating system kernel whose features are summarized in section 2.2.

2.1 A Programming Language for Distributed Computer Systems

The programming language EPL [13] was selected for use in this

study for several reasons. It is representative of the types of experimental languages being proposed for distributed computer systems. As such, it has a sufficient number of the constructions needed to decentralize software. It is particularly well suited to experimentation since it makes no assumptions about data types and structures nor does it place any constraints on the relationships between concurrent processes. For example, it is possible to construct pipelines of processes, co-routines, conventional procedures, data-flow structures, etc.

The programming language EPL provides a single abstract framework for defining concurrent programs -- the act. Instances of acts, called actors (more commonly termed processes), may be created dynamically. All actors are concurrent. Acts may be defined parametrically, so that multiple instances of actors (processes), defined by the same act, may be provided with different initial values. Actors may terminate only their own activies. This completely general structure is intended to encourage the use of concurrency to refine programming abstractions, just as procedures are used to construct layered architectures for sequential software.

The data spaces of actors are independent prohibiting the sharing of variables among actors. Actors are allowed to interact only by transmitting messages that are copied from the sender's data space to the receiver's data space. This restriction helps to make EPL programs independent of the interconnection structure of a distributed computer system. Transmission of messages is not implicitly buffered. Thus, the interprocess communication mechanism can also be used to synchronize actors. A sending actor is delayed until a receiving actor is prepared to accept the message. Also, a receiving actor is delayed until a sending actor is prepared to transmit a message.

The message transmission mechanisms are also the basis for a nondetminstic control structure. Receiving actors may elect to receive from a specific source, or to receive from any source. In the latter

case, the choice of a sending actor is nondeterministic.

Figure 2-1 shows how these concepts can be used to define an implementation of the Sieve of Eratosthenes. This algorithm identifies successive prime numbers by determining if successive integer are divisible by any prime that has been found. The code fragement shown implements this algorithm by defining a number of actors; one for each prime number that has been found. Successive integers are communicated from actor to actor unless they are found to be divisible by an existing prime. If an integer is not divisible by any existing prime, then the integer is assigned to its own actor as a new prime number. A single act is used to define all of the actors needed by this algorithm. The sieves function as a pipeline of actors and can be applied to a stream of integers. Each actor can potentially function in parallel with other actors defining the pipeline and can be distributed to a separate processing unit of a multi-computer system. This approach should be contrasted with a centralized implementation in which an array would be used to record prime numbers and a iterative construction would be used to compare successive integers to the set of prime number that have been found. The centralized version can be expressed as a single sequential process.

2.2 Run-Time Environment

The run-time environment of an EPL program must provide functions that create and delete processes, that schedule processes for execution, and that transmit messages between actors. The implementation of the last function realizes the non-deterministic control structure of EPL. The following paragraphs describe these functions and their implementation by an operating system kernel for a single CPU, LSI-11 microcomputer. This system was used to prototype decentralized algorithms written in EPL and to collect the initial performance data describing their use of the kernel.

The EPL compiler emits reentrant code. The scope of variables is such that the total data space required by an actor (process) may be

computed at compile time. These language features make it possible to implement a simple run-time storage management policy. When a new actor is created, space for its local variables is allocated as a contiguous set of locations from an available free pool of space. Operands are always addressed relative to the base of this area. An actor (process) descriptor record is prefixed to the data area for each actor. No attempt is made to recover this space when an actor terminates.

Actors that are logically enabled to run are linked in a circular chain that defines a ready list. A non-preemptive scheduling algorithm is used to assign the CPU to one of the ready actors. An additional language feature of EPL allows an actor to voluntarily relinquish its turn in the ready chain before the occurence of some condition that logically blocks its progress (e.g., sending a message to a actor that is not ready to receive).

Figure 2-2 illustrates how descriptors are linked to define the ready chain and lists of actors that are logically blocked. In figure 2-2, actors A, B, and C define the current ready chain. Actors D, E, and F are blocked waiting to send messages to actor B. Their descriptors have been removed from the ready chain and form a list associated with the descriptor of the destination actor. Actors G and J are blocked waiting to receive messages and are removed from the ready chain. The actor G is waiting for a message from a specific source other than H or I. The actor J is waiting to receive a message that has not been sent.

The operating system kernel of EPL defines six functions that manipulate these lists to manage actors and their interactions. These functions are

CREATE space for a new actor

RUN an actor

SEND a message

RECEIVE a message

TERMINATE an actor

SYSTEM call

The functions CREATE, RUN, and TERMINATE supervise actor creation and and deletion. The functions SEND and RECEIVE implement interprocess communication and synchronization. The SYSTEM call allows an actor to relinquish its turn in the ready chain.

The function CREATE allocates space. The function RUN transmits parameter values from the parent to the child actor and includes the child in the ready chain. Both functions are needed to instanciate a new actor. The function TERMINATE permanently removes an actor from the ready chain. The function SEND will block the running actor or it will copy a message between data spaces and enable a suspended receiving actor. The function RECEIVE will block the running actor or it will copy a message between data spaces and enable a suspended sending actor. The choice between a specific source and any source is determined by a parameter of this system call. The only context that must be saved by these system calls is the program counter and the register defining the base of the data area of the running actor.

3 PERFORMANCE DATA

Performance was measured by counting executed instructions. This was accomplished using the breakpoint trap of the LSI-11. Instruction counts exclude the instructions executed by software implementation of common arithmetic instructions such as multiply and divide. These standard arithmetic operations were counted as one executed instruction.

Table 3-1 gives the execution times of each of the six kernel functions used by EPL programs. These cost figures are consistant with values reported for similar kernel designs [15]. The cost figures are

very small and significantly less that those associated with general purpose operating systems kernels (e.g., UNIX). These small costs are explained by the use of a simple memory management algorithm, the lack of protection mechanisms, the use of non-preemptive scheduling algorithms, and low cost switching.

The primary objective of the performance measurements was to determine how frequently these operations were used by decentralized software. This would establish an initial estimate of the costs of decentralization. The kernel was modified to measure the number of instructions executed by EPL programs between successive calls to perform a function of the operating system kernel. These instruction sequence lengths were accumulated as a histogram by the kernel and reported at the end of execution of EPL programs. Six programs were executed to develop an estimate of how frequently the kernel functions were utilized.

Table 3-2 summarizes the observed characteristics of this performance measure for the programs sampled. The programs executed included the Sieve of Eratosthenes described in section 2.1, a cellular automata (Life [8]), and a two-player video game. These codes represent complete programs. In addition to these benchmarks, several smaller constructions were also exercised as benchmarks. These latter codes represent data structures and control constructions that would be used as building blocks in larger programs. They include a decentralized table structure for the Small Set of Integers problem discussed in [9], a limited implementation of guarded commands involving input and output statements similar to those proposed in [9], and and implementation of the eventcounter and sequencer synchronization mechanism proposed in [14]. The average length of instruction sequences executed between kernel calls is shown for each of these programs.

In general, the observed lengths of instruction sequences were very short and exhibited little variation. This claim is justified by the frequent occurence of lengths of 20 or less instructions. Figure 3-1

illustrates the distribution of observed lengths for the first three programs of table 3-2. None of the programs sampled executed more than 17,000 instructions between successive kernel calls. The longest message text transmitted was 9 words. In general, messages were the order of 2 words or less. These short message lengths are probably due to the use of structures of actors to represent single data structures (as in figure 2-1). Because each actor represents a very elemental unit of information, communications with its environment are limited.

Assuming that concurrent programs running in steady state are primarily using the kernel to send and receive messages, more than 2/3 of the instructions executed by the sampled decentralized software are executed by the kernel. When compared to a functionally equivalent, centralized implementations, these instructions represent an overhead associated with decentralization. For example, the Sieve of Eratosthenes would normally be written as a sequential program requiring no support from an operating system kernel.

This overhead represents the cost of decentralization, and is a basic trade-off that must be considered when using a distributed system to achieve goals such as enhanced reliability or incremental growth. These figures are mitigated by the parallelism that is inherent in many of these decentralized software structures. Because multiple computers will be used to execute decentralized software, performance improvements due to parallelism will offset these overheads. This consideration futher complicates analysis of the trade-offs of distributed computing.

An initial study [2] has attempted to clarify the extent to which performance improvements due to parallelism offset the implementation costs for such decentralized software structures. The results of this study were limited to a specific data structure organized as a pipeline of actors similar in organization to the code shown in figure 2-1. Queueing models were used to compare the performance of functionally equivalent decentralized and centralized information structures under similar loads. These models indicate that for moderate to heavy usage

of a shared data structure, the decentralized implementation should exhibit comparable performance to that of a centralized implementation, assuming implementation costs are similar to those shown in table 3-1. These preliminary models suggest that decentralization, which depends on the use of concurrency (and its associated implementation costs), can be effectively implemented at suprisingly low levels using conventional computer technology.

4 SUMMARY

The performance data of section 3 demonstrates that although the cost of individual kernel functions is low, they are used very frequently by decentralized software. Therefore, implementation costs represent a potentially large overhead in executing decentralized software. There are preliminary indications that these high implementation costs are offset by performance improvements due to the inherent parallelism of decentralized software. Thus, the overall performance of a decentralized implementation of an algorithm may be comparable to a functionally equivalent centralized implementation

These observations suggest an objective of performance models for decentralized software be explainations the trade-off between performance improvements due to parallelism and the implementation costs of that parallelism. Such modelling results could then be used to directly compare the performance of decentralized software with that of fuctionally equivalent, centralized software. Such comparisons would provide a basis for understanding the costs of distributing computer applications to achieve other attributes (e.g., reliability).

Initial efforts at modelling the behavior of EPL programs have employed queueing networks [2] and probabilistic grammars [15] to model events that correspond to the transmission of messages between actors. Continued application of such modelling methods appears fruitful. Successful applications of these models will require a performance measurement facility that can provide data needed to define model parameters

values and to validate models.

Figure 4-1 illustrates a research facility designed to provide such support. The facility is hosted by a PDP-11/60. Five LSI-11s define a losely-connected, multi-computer subsystem that can be loaded and observed by the host computer. The multi-computer subsystem has an interconnection structure that can be varied to study different topologies. The host computer can be used to develop software for execution by the multi-computer subsystem. It can also be used to record, reduce, and analyze performance measurements describing the behavior of the multi-computer subsystem. An alternative role of the host computer is the simulation of an environment for the subsystem of LSI-11s.

This research facility makes it possible to measure performance of decentralized software using a truely parallel system. The measurements cited in this paper were obtained for a single CPU implementation of EPL programs. The single CPU implementation of EPL is being revised to control the subsystem of LSI-11 computers. The objective of the design is to demonstrate implementation independence of EPL programs by executing unmodified programs using a variable number of processing elements. Each computer will have a kernel that provides an identical set of functions to support the execution of and EPL program. The approach is based on a kernel design that maintains partial state information describing only those actors executed by its processor. This design introduces a number of additional scheduling issues such as the allocation of actors to computers, and global versus local memory management policies.

The implementation of the EPL kernel for a multi-computer system is expected to provide an estimate of the implementation costs for a distributed computer system programming language. These costs are expected to be somewhat larger than those cited in section 3 for several reasons. The use of slower, possibly indirect, computer interconnection structures will introduce delays in the transmission of messages between actors. More important, decentralization of the kernel will require the

use of low-level communications to coordinate the activities of instances of the kernel. These latter costs are not present in the single CPU implementation of EPL since the kernel has timely access to all state information describing actors. However, it is expected that when more sophisticated software applications are implemented and measured, these increased costs will be offset by the performance improvements due to the parallelism of longer instruction sequences between kernel calls.

5 REFERENCES

[1] Atkinson, R.R., and C.E. Hewitt, "Specification and Proof Techniques for Serializers", IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, Jan. 1979, pp10-23.

[2] Balkovich, E.E., and C. Whitby-Strevens, A Model and Measurements of a Decentralized Information Structure, Department of Electrical Engineering and Computer Science, University of Connecticut, (in preparation), 1979.

[3] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol. 21, No. 11, Nov. 1978, pp. 934-941.

[4] Chang, E., and R. Roberts, "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors", Communications of the ACM, Vol. 22, No. 5, April 1979, pp. 281-283.

[5] Dijkstra, E.W., "Self-Stabilizing Systems in Spite of Distributed Control", Communications of the ACM, Vol. 17, No. 11, Nov. 1974, pp. 643-644.

[6] Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Communications of the ACM, Vol. 18, No. 8, Aug. 1975, pp. 453-457.

[7] Feldman, J. "High Level Programming for Distributed Computing", Communications of the ACM, Vol. 22, No. 6, June 1979, pp. 353-368.

[8] Gardner, M., "Mathematical Games", Scientific American, Vol. 223, No. 10, Oct. 1970, pp. 120-123.

[9] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, Aug. 1978, pp. 666-677.

[10] Kant, R.M., and T. Kimura, "Decentralized Parallel Algorithms for Matrix Computation", in Proceedings of the 5th Annual Synposium on Computer Architecture, Palo Alto, CA, April 1978, pp. 96-100.

[11] LeLann, G. "Distributed Systems - Towards a Formal Approach", in Proceedings of the IFIP Congress, August 1977, pp. 155-160.

[12] May, M.D., R.J.B. Taylor, and C. Whitby-Strevens, "EPL - An Experimental Programing Language", in IEEE Conference on Trends and Applications: Distributed Processing, Gaithersburg, MD, May 1978, pp. 69-71.

[13] May, M.D., and R.J.B. Taylor, The EPL Programming Manual Distributed Computing Project, Report No.7, Dept. of Computer Science, University of Warwick, Coventry, England, 1979.

[14] Reed, D. P. and R. K. Kanodia, "Synchronizations with Evencounters and Sequencers", Communications of the ACM, Vol. 22, No. 2, February 1979, pp. 115-123.

[15] Whitby-Strevens, C. "Towards the Performance Evaluation of Distributed Computing Systems", in Proceedings of COMSAC 78, Chicago, IL, Oct. 13-16, 1978, pp. 141-146.

[16] Wirth, N., "Design and Implementation of Modula", Software: Practice & Experience, Vol. 7, 1977, pp. 67-84.

Figure 2-1

Sieve of Eratosthenes

```
// This ACT is a sieve which is initialized to the first number sent
// it, n.  It sieves out all multiples of n that are subsequently
// sent.  All numbers that are not sieved out are sent to another
// sieve that it creates within itself, in sequence.

CONST sieve = ACT [] :                  // define the ACT sieve
  REC n THEN                            // receive the first message
    (                                   // defining n
    ACTOR nextsieve : [] :: sieve;      // create another sieve
    REP                                 // repeatedly receive and
      REC p THEN                        // sieve integers
        IF (p REM n) = 0 THEN SEND nextseive : [p] FI
    PER
    );

ACTOR firstsieve : [] :: sieve;         // create the first sieve
LET m = 2;
REP                                     // send successive integers
  SEND firstsieve : [m];                // to the first sieve
  m := m + 1
PER
```

Figure 2-2

Kernel Data Structures

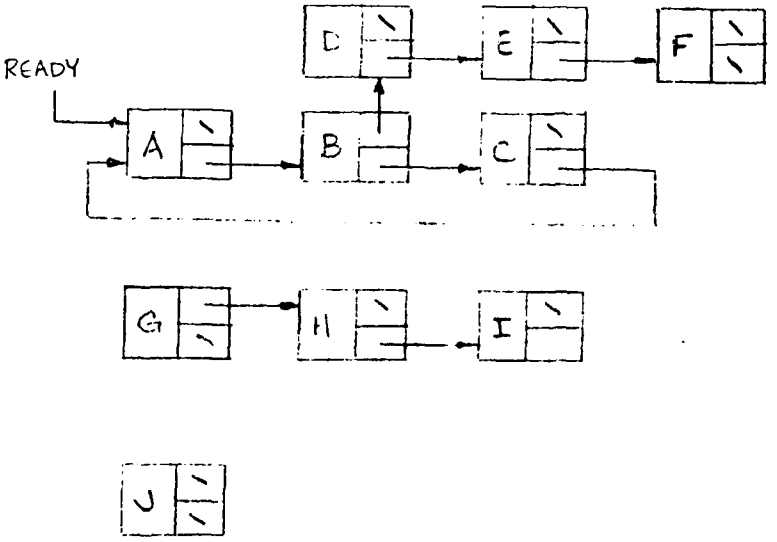


Table 3-1

Execution Times of Kernel Functions

Kernel Function	Average No. of Executed Instructions
CREATE	17
RUN	16
TERMINATE	10
SEND	20+n*
RECEIVE	20+n*
SYSTEM	6

*n = length of the message

Table 3-2

Use of Kernel Operations by EPL Programs

Program Name	Ave. Ln. of an Inst. Seq.	Prob. of a Seq. of <= 20 Inst.
Cellular Automata	11.3	0.89
Sieve of Eratosthenes	7.5	0.99
Video Game	7.9	1.00
Small Set of Integers	9.2	1.00
Eventcounters & Sequencers	8.5	0.97
Guarded Commands	5.8	0.98
Average	8.4	

Figure 3-1

Distribution of Observed Lengths of Instruction Sequences

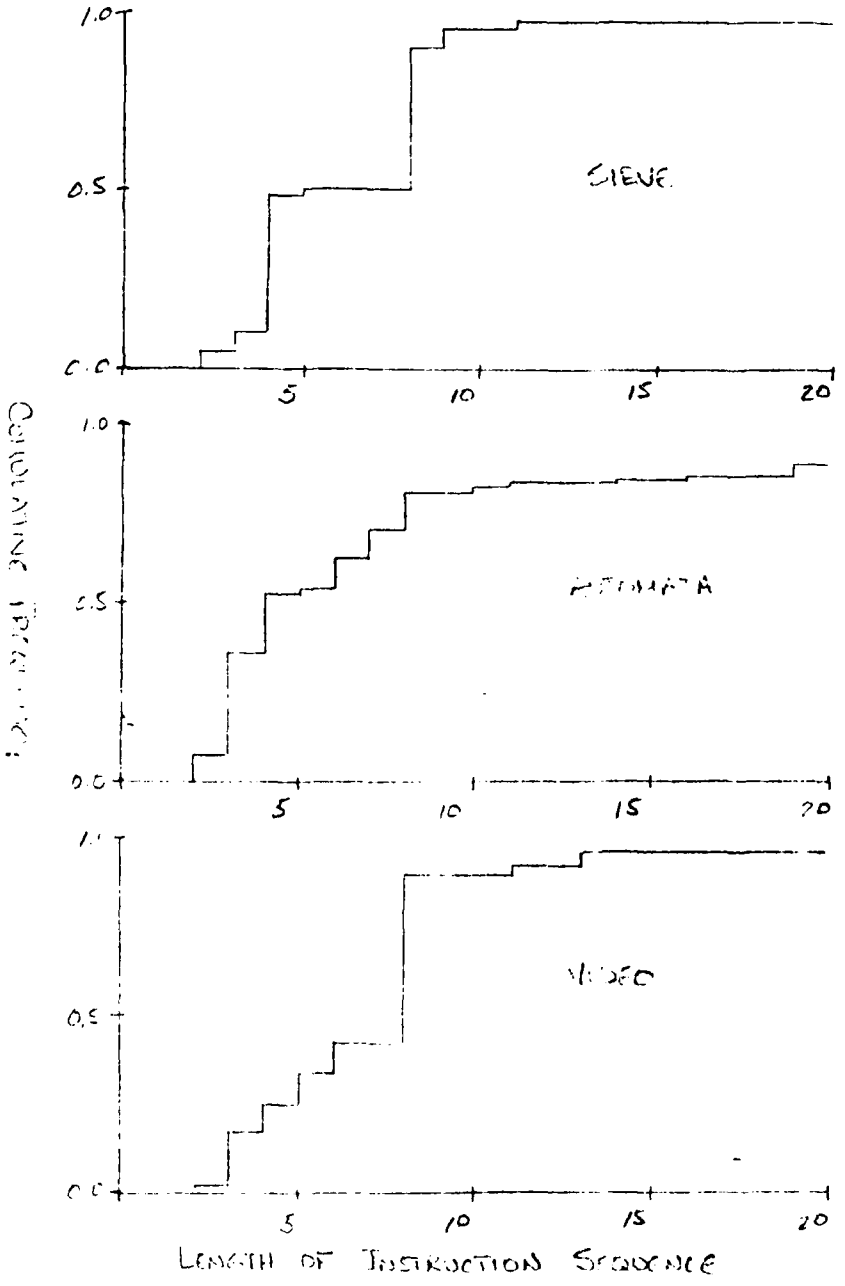
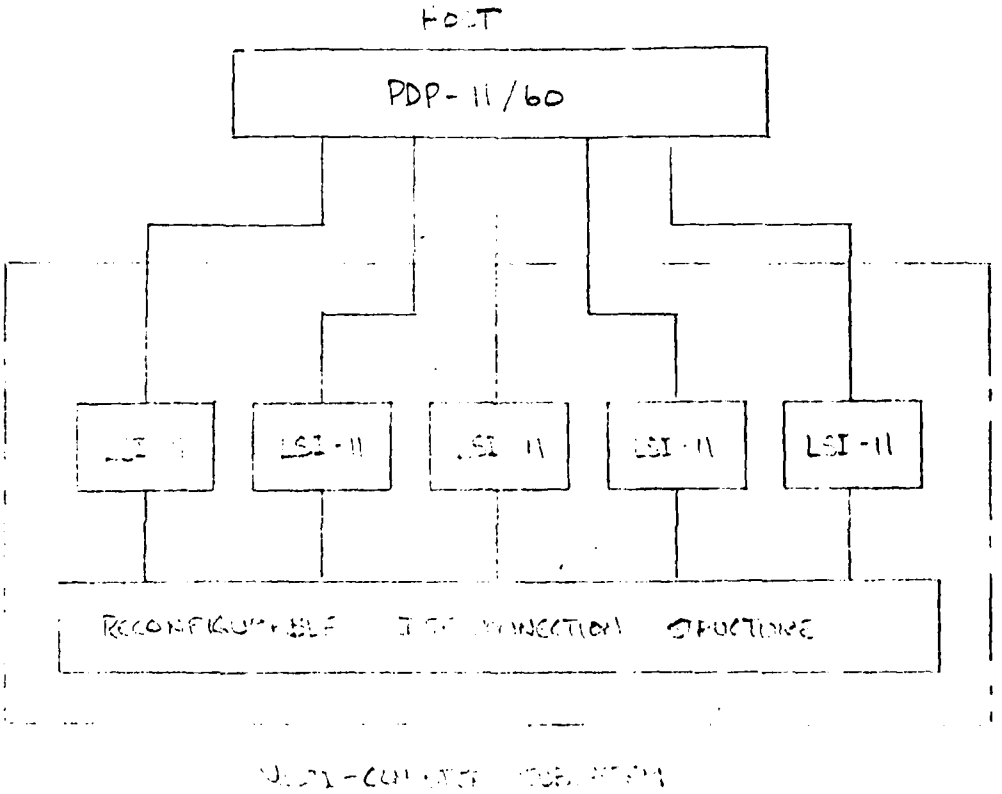


Figure 4-1

Distributed Computer System
Performance Measurement Facility



Part VI

Performance of Distributed Software Implemented by a Contention Bus

by

E. Balkovich and J. Morse

ABSTRACT

The dominant features of software for distributed computer systems are communication between processes and potential parallel execution of processes. This paper examines how such a software architecture can be expected to use a distributed computer system based on a contention bus. Processes tend to be fine-grained and transmit short messages. An operating system kernel that implements process interactions introduces an additional source of short messages used to coordinate process interactions. Existing analyses of contention bus performance show that such traffic patterns will be unable to utilize more than a fraction of the potential contention bus bandwidth. An additional concern is that many important assumptions made by these models are violated by such an implementation of software. This paper suggests some possible ways to extend performance models to more accurately reflect the features of software for distributed computer systems. It also suggests an approach to improving channel efficiency for implementations of such software.

KEY WORDS

Distributed Computing, Distributed Software, Contention Bus, Shared Bus, Performance Evaluation, Queueing Theory, Operational Analysis

1 Introduction

Typically, the total hardware resources of a distributed system are implemented by several autonomous computers that are linked by a communication network. It is also necessary to distribute software that implements system-wide services and functions in a similar way. One approach to logically distributing or decentralizing the functions provided by software is to implement software as a collection of independent processes that cooperate to provide a service. One objective of such designs are systems that continue to operate in the presence of failures. This approach has been labeled as a task force [Jon79], or a system of actors [Hew77] or objects [Gol80]. Generally, the component processes that implement a function are executed by different computers and interact via messages rather than shared variables. How this software architecture uses hardware raises a number of important performance questions.

The decomposition of a function into multiple, concurrent processes may allow the inherent parallelism of a distributed system to be exploited in implementing a function. On the other hand, the required interprocess communication, particularly when it uses the communication network, can introduce delays. The tradeoff between these two factors is impacted by design decisions that determine the decomposition and the allocation or assignment of processes to the computers of a distributed system. These interactions are further complicated by the characteristics of the communication network linking the computers. For example, the performance of a contention bus is sensitive to load characteristics and requires the careful design of protocols that support process interactions.

There is a clear need to model both the potential parallelism and the communication traffic generated by a decentralized software design. This information is needed to define the input parameters of optimal task allocation algorithms [Chu80]. It also provides a basis for selecting low-level algorithms that control hardware interconnection

structures. For example, time-division multiplexing and backoff algorithms provide optimal utilization of a shared communication channel under very different load assumptions.

This paper reports on the author's experience designing a local-area network to support a decentralized software architecture. The interconnection structure is a contention bus similar to the Ethernet [Met76]. The distributed software architecture is similar to that suggested in [Hoa78].

The following sections review the features of a distributed software architecture and briefly summarize the design of a prototype computer system to support experimentation with such software. Preliminary experience with this prototype system suggests that only a few of the existing models of contention bus performance can be applied to understanding the system's performance. The primary objective of this paper is to review the models of contention bus performance and to discuss the interpretation of their results for the system being designed. The applicable results suggest some algorithms for using the contention bus. The summary discusses these algorithms and the author's views on the performance models that will be needed to completely understand the behavior of the system.

2 Background

The nature of the primitive programming concepts needed to support a decentralized software architecture is a subject of current research. Most of the results in this area [BrH78, Coo80, Dod80, Fel79, Hoa78, Lis79, May78, Mao80] propose programming languages tailored to distributed computing. In spite of the diversity of results, several common themes emerge. These common ideas provide insight into the factors that determine the performance of distributed systems.

One feature of distributed software is the use of autonomous, concurrent processes as a basic unit of software modularity. Autonomy is usually taken to mean that the behavior of one process cannot forcibly

influence the behavior of another process. Usually, this is achieved by preventing sharing of variables and by placing few, if any, restrictions on the relationships of processes. In the extreme, the notion of a concurrent process can be used as the only source of program modularity [BrH78, Hoa78, May78], replacing conventional concepts such as subroutines. Since processes are concurrent, the resulting software structures may have a significant potential for parallelism.

Without shared variables, process interactions are implemented with messages. In many language proposals, messages play two roles: they provide a mechanism for transmitting data, and they also provide the only mechanism for synchronizing processes. Given this latter use of messages, and the extensive use of fine-grained processes, interprocess communication is the dominant feature of decentralized software. Since processes are generally resident at distinct computers, the hardware interconnection structure will generally be used to implement message-based interactions of processes.

The discussions that follow are based on the design and implementation of an experimental system to support decentralized software with these features [Bal80b, Fon80]. Distributed software is written for a virtual system with an unlimited number of independent virtual machines implemented by an operating system kernel. No distinction is made between process interactions occuring on the same physical machine and process interactions involving multiple computers. An operating system kernel at each machine provides identical virtual machine functions. The implementation of these functions is distributed by a partition of the process state information. Each instance of the kernel is responsible for only the set of processes executed by one physical machine. Process interactions that involve processes on other physical machines require that multiple instances of the kernel cooperate to achieve the interaction. This cooperation is accomplished by exchanging kernel-level messages transmitted over a contention bus similar to the Ethernet [Met76]. Figure 1 illustrates the layers of software used in the imple-

mentation. The p[i,j] represent individual processes used to distribute

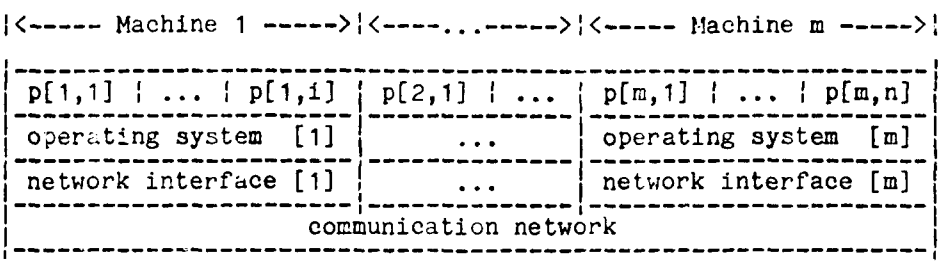


Figure 1
Layers of the Distributed System

software.

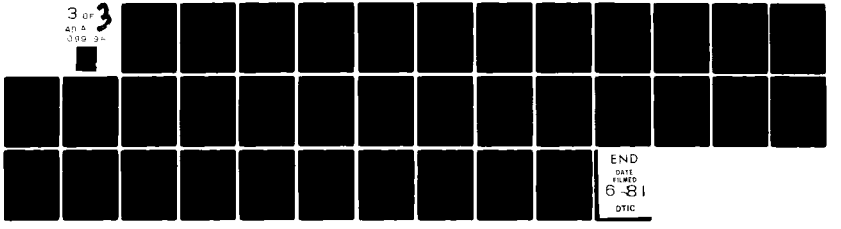
The operating system kernels [Fon80] implement a virtual machine that supports the EPL programming language [May78, May79]. Interprocess communication is accomplished with messages that are transmitted synchronously. Generally, communication takes place between named processes, although there is a provision for nondeterministic selection of a message source. Most applications of the language can be characterized as using large numbers of small, frequently interacting processes that transmit short messages. The architecture of these programs most closely resembles that of CSP [Hoa78].

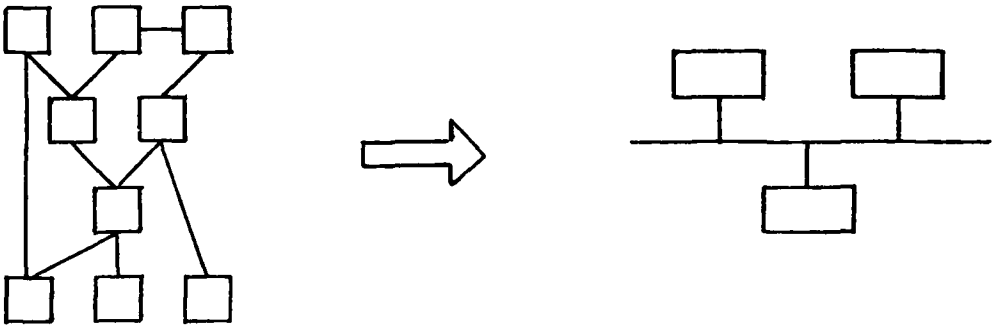
3 Structure of a Performance Model

The dominant characteristic of a distributed software architecture is communication between processes that may be executed in parallel. Our initial goal is to understand the performance of specific decompositions and assignments of processes to hardware linked by a contention bus. Figure 2 illustrates the major components of a such an explanation.

AD-A099 195 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
DECENTRALIZED SYSTEMS.(U)
DEC 80 E E BALKOVICH DAS660-79-C-0117
UNCLASSIFIED TR-CS-15-80 NL

3 OF 3
AD A 099 195





Logical Structure of Software | Operating System Kernel | Physical Structure of Hardware

Figure 2

Model Components

The software architecture can be described in terms of a logical interconnection structure of autonomous processes. Conceptually, the performance of the software is determined by the characteristics of message traffic between processes and the computing required to respond to and generate messages. The operating system kernel maps or projects the logical structure of the software onto the physical structure of the hardware. It does so by multiplexing a single computer to execute several processes and by sharing the contention bus to implement multiple logical channels between processes.

The performance problem can be reduced to explaining the performance of the physical system executing the software structure. Conceptually, the performance of the physical system is determined by the characteristics of the contention bus, the characteristics of the mes-

sage traffic presented to it by each computer, and the composite computing load presented to each computer. These characteristics must be derived from features of the logical structure of distributed software and the mapping or projection of that logical structure onto hardware.

The logical structure of the software defines a partial ordering of message traffic that arises from the synchronization of processes to achieve correct behavior. This message traffic can be characterised by distributions of message lengths and time intervals between transmissions. The operating system and the contention bus map this ordering and these distributions into a totally ordered traffic pattern with different characteristics.

The allocation or assignment of processes to computers aggregates logical message traffic into a single stream of traffic presented to the contention bus by a computer. The characteristics of this physical message traffic are modified by allocation decisions that allow processes resident on the same computer to interact without using the communication network. Allocation and scheduling decisions have a corresponding effect on the computational requirements for a computer.

A source of significant perturbations in the traffic characteristics presented is the introduction of protocols used by the operating system kernel to implement process interactions. Generally, these protocols modify the arrival pattern of messages and alter the distribution of message lengths by introducing an additional source of short messages. Clearly, the delays and utilization of communication network will depend on the characteristic usage of the network. In turn, the performance of the contention bus plays a role in determining usage, since the entire system is closed.

These arguments suggest that as a minimum, it is necessary to model usage of a contention bus by particular software architectures and their potential mappings onto the hardware. Most models of contention bus performance assume that such information is available. The following

section reviews these models and summarizes the results that are relevant to understanding how the architecture of decentralized software will use a distributed computer system based on a contention bus. These results suggest some operating system algorithms that may improve the use of the contention bus. These alternatives are discussed in the summary of this paper. Developing a suitable model of the software architecture still represents a major challenge in understanding distributed systems. Although the detailed formulation of a closed model of a distributed system remains an open question, approaches to formulating such a model are discussed in the next section.

4 The Network Model

Our experimental distributed computer system assumes a communication network implemented as a single channel that is shared by all computers. Such a channel can be implemented as a radio broadcast link, or as a wire. In either case, computers that wish to transmit over the channel must contend for use of the shared channel, since only one computer at a time may use the channel.

4.1 History of Analytic Models

A primary measure of contention bus performance is throughput efficiency. Throughput efficiency is calculated as the ratio of the total available bandwidth of a channel and the maximum bandwidth that can be used for data communication. The most important assumption made in many analyses is that message sources may be modelled as an infinite number of identical, poisson processes which together provide a constant offered load lambda to the channel. The parameter lambda is expressed as some fraction of the total channel capacity. Generally, it is also assumed that the time to transmit one message is a constant tau.

The basis for analytic models of contention bus performance can be traced to the Aloha radio network [Abr70]. The Aloha network assumes that the stations cannot know if the channel is busy. Whenever a station has a message to transmit, it will immediately transmit the mes-

sage. A given message, beginning at time T, will experience a collision if any other message begins transmission within plus or minus tau seconds of T. Thus the collision interval for messages of fixed size is 2 tau. It can be shown that the maximum efficiency of the channel is 1 / (2 e), or 0.1839. When the offered load reaches 0.1839, the remaining channel capacity can be attributed to damaged packets (due to collisions), or to idle time (due to no station being ready to transmit). In spite of the fact that the channel is fully 50% idle, increasing the offered load beyond 0.18 results only in more collisions, not more effective utilization of the channel.

The analysis that leads to the 1 / (2 e) limit on efficiency assumes that the arrival rate of new messages and the arrival rate of retransmissions due to collisions are both exponentially distributed. In practice, these assumptions are valid if (1) the mean time before retransmission is large relative to tau, and (2) not too many retransmissions occur. As long as the offered load stays below 0.15, these conditions are satisified.

A simple refinement of the Aloha scheme is to synchronize transmissions. Ideally, time is divided into slots lasting tau seconds (where tau is the time for one message as defined above). With this scheme, called "slotted Aloha", messages collide when they are transmitted in the same slot. The collision interval is reduced to tau. It can be shown that the maximum efficiency of the channel is now 1 / e, or 0.3679. Adding synchronization effectively doubles the capacity of the channel. In this case, when the offered load reaches 0.3679, the remaining capacity of the channel is consumed by collisions, and the channel is never idle [Lam79]. Again, the same assumptions about arrival time distributions invalidate the model when offered loads exceed 0.30.

Ethernet [Met76] introduced carrier detection circuitry at each station to determine when other stations are transmitting. As a result, stations can defer transmission when another station is found to be

transmitting. This avoids most of the collisions associated with Aloha. However, there is still the possibility of collision. Two stations can begin transmission during an interval of time so short that neither can detect the other in time to defer. The deferral mechanism tends to synchronize message transmissions to the completion of messages, particularly under heavy loads. For this reason, Ethernet tends to behave like slotted Aloha even though there is no explicit slot synchronization.

Adding deferral and collision detection reduces the collision interval from tau to some fraction of tau, depending on the message size. If messages are long relative to the slot time (the interval required to establish a detectable carrier signal), then very high channel utilization is possible. If a message does not collide in the first slot time, then the channel will remain clear for the duration of the transmission. Thus, there are two kinds of slots -- slots during which contention can occur (the first slot of every message), and clear slots (subsequent slots of messages that experience no collisions). The channel efficiency for contention slots remains at a maximum of 1 / e, but the clear slots have efficiency of 1.0 -- they can always be used to successfully transmit data. The resulting throughput efficiency for messages of 4096 bits (85 slot times) is shown to be 0.98 [Met79].

The retransmission policies of Aloha and slotted Aloha assume that the delay is "large". The Ethernet model assumes that after a collision, each station that is ready to transmit will do so in the next time slot with probability 1 / Q (where Q is the total number of stations ready to transmit). It can be shown that this much stronger assumption satisfies the conditions of Abramson's model [Abr70], so that the 1 / e upper bound on efficiency still holds. The advantage of the stronger Ethernet assumption is that it becomes possible to predict the delay characteristics of the channel. Delay is defined as the amount of time a station has to wait after the channel becomes ready, but before it successfully acquires the channel. An accurate model of channel delay is essential for evaluating the use of a contention bus in implementing

the interaction of software processes.

A number of key points in the above discussion are applicable to our model of distributed computing. The usable throughput of a shared channel will be a fraction of the total capacity of the physical link. This fraction will depend on the length of messages transmitted. Under heavy loads, there will be many collisions and consequent retransmissions which will impact throughput efficiency and transmission delays. Finally, the analysis is based on a set of restrictive assumptions about arrival rates and service rates that may not be satisified by a distributed computer system.

4.2 Applying the Analysis

In order to apply these analytic models to our system, it is necessary to identify where the system violates assumptions made by the models. In our system, there is a finite number of message sources, and in many cases this number will be small. This violates the infinite source assumption. The software processes that generate messages do so to effect cooperation. In many cases, the processes are synchronized and violate the assumption of independent sources. Most of the messages will be used for synchronization and control of processes. These messages are very short and do not satisify the bounds on message length that lead to high channel efficiency. Processes form pairs and engage in extended dialogues to carry out higher level protocols. The result is that message arrivals tend to be "bursty" and violate the assumption of exponential interarrival times.

The implementation of Ethernet, as specified in [Dig80], differs in a number of important ways from the "ideal" model [Met76]. The strategy for re-transmission after collision is implemented by a backoff algorithm. The backoff algorithm does not use Q to set the probability of re-transmission in the following slot. A single station in a decentralized implementation cannot know Q. The backoff algorithm provides a method of estimating Q. Analysis shows that the backoff algorithm's es-

timate of Q is not very good unless collisions occur between only two stations. The adequacy of the backoff algorithm therefore depends on the assumption that the majority of collisions only involve two stations. It has been shown by simulation [Mar80] that this assumption generally holds as long as broadcast messages do not require responses from all stations.

For software architectures such as those assumed in our work, the infinite source assumption must be replaced by a model of n sources (where n, the number of computers in the system, is not only finite, but in many cases quite small). The performance of Ethernet for a limited number of stations has been analyzed [Alm79, Met76]. These studies show that for n >= 10, the finite source model closely approximates the infinite model. For n < 10, Ethernet efficiency is, in all cases, better than that predicted by the infinite source model.

For our software architecture, the assumption that stations are independent is surely false. Since processes must synchronize in pairs in order to transmit information, there is a tendancy for computers to "pair up" while a protocol is carried out by the operating system kernel. It is also expected that many distributed algorithms will exhibit a tendancy for pairs or groups of processes to engage in extended dialogues. For significant lengths of time, channel usage will be dominated by small subsets of computers. There are preliminary indications that this pattern of usage improves the efficiency of an Ethernet-like contention bus, but more work remains to be done on this subject.

A major concern about contention bus performance for our assumed software architecture is the preponderance of short messages. If Ethernet parameters [Dig80] are used for illustration, then the high bandwidth (10 megabits) and long slot time (60.8 microseconds) cause any message of less than 46 bytes (excluding address information) to fit into a single slot. Our analysis indicates that for many applications all, or almost all, messages will fit in a single slot. As an example, a typical protocol to transfer a small block of data between 2 processes

requires 4 messages [Fon80]. The first 3 message are each 8 bytes in length, and the fourth is 8 bytes plus the length of the data block. Thus, the high potential efficiency of Ethernet will not be achievable by our assumed software architecture. Efficiency will be no better than slotted Aloha since every slot is a potential contention slot. The efficiency of the Ethernet, used for our purposes, would be 1 / e of the channel capacity, or about 3.6 megabits per second. Whether this would provide adequate performance remains an unresolved question.

Some preliminary insight into the kind of message traffic generated by a distributed software architecture has been gained by experimenting with the programming language EPL [Bal80a]. Typical processes execute a very small number of instructions (almost always less than 20) between calls to the operating system for message transfers between processes. The result is very short, very frequent messages. This message traffic cannot be accurately characterized as exponential. The pairing of processes and the bursts of messages generated to synchronize processes and coordinate data transfer suggests that the arrival pattern may be hyper-exponential. It is tempting to assume that the exponential arrival assumption may still be used to place a lower bound on expected communications delay, but this hypothesis is in fact invalid, as shown in [Cha78].

4.3 Future Models

The specific requirements of a distributed software architecture suggest a number of ways in which existing models of contention bus communication should be extended. Message traffic is self-limiting. Typically, processes generate most messages in response to messages from other processes. If messages are delayed by high loads on the channel, then processes will be blocked and will generate no further traffic until the current demand is satisfied. When this happens, the operation of the entire system is limited by the capacity of the communication network. On the other hand, if a channel has sufficient capacity to immmediately service transmissions, then the operation of the system is

limited by the processing speed of each computer. The system becomes "processor bound" instead of "I/O bound". Models should take both kinds of behavior into account. Locating the crossover point is important since it is expected that the performance characteristics of the two cases will be very different. Part of our current work has attempted to establish this crossover point by trace-driven simulation of a few specific cases [Sou80].

An accurate model of the response (or delay) characteristics of a communication network plays an important role in predicting this crossover point and the behavior of the system as a whole. It is not sufficient to predict only the mean of the response time -- its variance is required as well. The Ethernet exhibits a form of "last in first out" behavior that results in a high variance of response time under heavy loads [Alm79]. Under such circumstances, the performance of distributed algorithms may be erratic or even un-predictable. One can imagine a mode in which the system alternates between periods of processing, in which messages accumulate during long periods of bus contention, and periods of communications, when messages are finally transmitted, but in an order that ensures that processes remain blocked until the last message is transmitted. Such behavior is undesirable since neither communication nor processing is efficient.

If a standard queueing theory approach is to be successfully applied to analyze response, then something more complex than an M/M/1 queue is required. An embedded Markov process has been used in at least one study [Tob79] to yield good estimates of channel delay given both a finite number of stations and variable message lengths. Another approach that may prove fruitful is to account for collision resolution in the model of the server, rather than in the model of the source (as was done in [Abr70]). Collisions and re-transmissions are reflected in a reduction in the service rate rather than an increase in the arrival rate. If the source model is assumed to be finite, an M/G/1//m queueing system results. The problem with either the embedded Markov process or

the M/G/1//m approach is that any attempt to incorporate more of the essential features of the problem into a model increases the computational complexity. In [Tob79] numerical methods were used to derive the most useful results, while most of the important results in [Alm79] were derived from simulation. These methods often require vast computational resources to yield good results. The other major problem with these approaches is that it is very difficult to incorporate a model of the software into the analysis. The embedded Markov process method looks promising and should provide an opportunity for further research.

One approach to modelling the architecture of distributed software operational analysis [Den78]. This approach assumes a finite set of n processes, allocated among m nodes. Each process is modelled as a message generator. The time of occurrence and the destination of messages are modelled as random variables. Once a process has generated a message, it sends it and enters a wait state until it receives a reply message. Each process needs to be characterized by the frequency of its message generation, and by a probability vector. This vector represents the probability that the next message generated by the process will be routed to a specific destination. These probabilities are equivalent to the visit ratios used in operational analysis.

Using this approach, the interaction of the computers in the system is emphasized, while the details of the channel are subsumed in the model as a simple queued device. This may be a useful model as a first order approximation. It can probably be applied to determine whether a specific system configuration will operate I/O bound (the contention bus is the saturated device) or processor bound (some process is the saturated "device").

5 Summary

Our review of contention bus performance models has led to several conclusions about their applicability to systems implementing distributed software architectures. Interconnection structures, like the Ether-

net, use detection principles that naturally divide the channel capacity into clear slots and slots during which contention may occur. When messages span many slots, very high utilization of the channel is possible under heavy loads. However, when messages are short, the utilization of the channel is limited to a fraction of that raw channel capacity.

Under appropriate assumptions these models are useful, but not entirely adequate for our purposes. In particular, the architecture of distributed software places demands on the channel that in many important ways violate assumptions made by the models. The number of sources (computers) is finite and generally cannot be expected to operate in an independent manner. Preliminary data suggests that distributed software will generate bursty message traffic of predominately short messages. This does not satisify the assumptions made about the distributions of traffic patterns.

These observations suggest that a contention bus should always be sized so that it is lightly loaded. Given that this is not practical, then the communiction network becomes a bottleneck and a significant amount of message queueing will occur. Under these circumstances, it may be feasible to employ algorithms in the operating system kernel that extend the length of messages transmitted over the network. One approach that is being considered is to provide bulk service when transmitting messages. This approach would frame queued messages and collectively transmit them as a single, longer message. Implementation can be accomplished either by broadcasting all pending messages, or by grouping pending messages according to their destinations. The former approach should yield longer messages, but requires that each computer of the system receive and decode all message traffic.

The use of bulk service methods should be judged by its impact on performance. This requires understanding two aspects of how the network would be used. The first issue is to determine the average message length that would be generated using a bulk service algorithm and whether it would significantly alter the performance of the communication

network. To be successful, such an algorithm must generate aggregate messages lengths longer than one slot. This would increase the efficiency of the channel. The second, and perhaps more important issue is the impact of a bulk service policy on response time. This requires evaluation of both the mean and variance of response time and can probably only be accomplished using simulation techiques.

Our plan for future work is to extend the existing models of interprocess communication to include performance models of the distributed system as a whole. We hope to extend some of the analytic models to account for the particular characteristics of the work load and service time distributions that we feel characterize distributed software. The preliminary work using trace-driven simulation has proved fruitful and will be persued further. Finally, we plan to explore operational analysis models that would address both the software and the hardware architectures.

REFERENCES

[Abr70] Abramson, N., "The Aloha System -- Another Alternative for Computer Communications," Proc. AFIPS 1970 FJCC, AFIPS Press, Montvale, N. J. (1970), 281-285.

[Agr77] Agrawala, A. K., Bryant, R. M., and Agre, J., "Analysis of an Ethernet-like protocol," Proc. Computer Networking Symposium, IEEE Computer Society and N.B.S., (Dec. 1977), 104-111.

[Alm79] Almes, G. T., and Lazowska, E., "The Behavior of Ethernet-like Computer Communications Networks," Proc. of the Seventh Symposium on Operating System Principles, (Dec. 1979), 66-81.

[Bal80a] Balkovich, E. E., and Whitby-Strevens, C., "On the Performance of Decentralized Software," Proc. of the 7th International Symposium on Computer Performance Modelling, Measurment and Evaluation, Toronto, Canada (May 1980), 173-180.

[Bal80b] Balkovich, E. E., The Design and Operation of an Experimental Facility for Distributed Computer System Research, Computer Science Technical Report CSR-80-8, Department of Electrical Engineering and Computer Science, University of Connecticut, Storrs, CT, (September 1980).

[BrH78] Brinch Hansen, P., "Distributed Processes. A Concurrent Programming Concept," Comm. ACM, 21,11 (Nov. 1978), 934-941.

[Cap79] Capetanakis, J. I., "Generalized TDMA: The Multi-Accessing Tree Protocol," IEEE Trans. on Comm., COM-27,10 (Oct. 1979), 1476-1484.

[Cha78] Chandy, K. M., and Sauer, C. H., "Approximate Methods for Analyzing Queueing Network Models of Computer Systems," ACM Computing Surveys, 10,3 (Sept. 1978), 281-317.

[Chu80] Chu, W. W., et al., "Task Allocation in Distributed Data Processing," Computer, 13,11 (Nov. 1980), 57-69.

[Coo80] Cook, R. P., "*MOD - A Language for Distributed Programming," IEEE Trans. on S. E., SE-6,6 (Nov. 1980), 563-571.

[Den78] Denning, P. J., and Buzen, J. P., "The Operational Analysis of Queueing Network Models," ACM Computing Surveys, 10,3 (Sept. 1978), 225-261.

[Dig80] Digital Equipment Corp., Intel Corp., and Xerox Corp., The Ethernet, A Local Area Network, Data Link Layer and Physical Layer Specifications (Sept. 1980).

[DoD80] Reference Manual for the Ada Programming Language, United States Department of Defense, July 1980.

[Fel79] Feldman, J., "High Level Programming for Distributed Computing," Comm. ACM, 22,6 (June 1979), 353-368.

[Fon80] Fontaine, S. C., A Distributed Computer Operating System Kernel, Masters Thesis, Department of Electrical Engineering and Computer Science, University of Connecticut, Storrs, CT, (December 1980).

[Gol80] Goldberg, A., et al., Smalltalk: Dreames and Schemes, Xerox Corp., 1980.

[Hew77] Hewitt, C., "Viewing Control Structures as Patters of Passing Messages," Artificial Intelligence, 8, (1977) 323-364.

[Hoa78] Hoare, C. A. R., "Communicating Sequential Processes," Comm.

ACM, 21,8 (Aug. 1978), 666-677.

[Jon79] Jones, A. K., and Schwans, K., "TASK Forces: Distributed Software for Solving Problems of Substantial Size," Proceedings of the 4th International Conference on Software Engineering, (Sept. 1979), 315-330.

[Kle75] Kleinrock, L., and Lam., S. "Packet Switching in a Multiaccess Broadcast Channel: Performance Evaluation," IEEE Trans. on Comm,, COM-23,4 (Apr. 1975), 410-422.

[Kle78] Kleinrock, L., and Yemini, Y., "An Optimal Adaptive Scheme for Multiple Access Broadcast Communication," Proc. 1978 International Computer Conference, (1978), 7.2.1-7.2.5.

[Lam79] Lam, S., "Satellite Packet Communications -- Multiple Access Protocols and Performance," IEEE Trans. on Comm., COM-27,10 (Oct. 1979), 1456-1466.

[Lis79] Liskov, B., "Primitives for Distributed Computing," Proceedings of the 7th Symposium on Operating System Principles, (Dec. 1979), 33-42.

[Mao80] Mao, T. W., and R. T. Yeh, "Communication Port: A Language Concept for Concurrent Programming", IEEE Trans. on S. E., SE-6,2 (March 1980), 194-204.

[Mar80] Marathe, M., "A Study of NI Architectural Alternatives" Digital Equipment Corp. internal report, (Aug 1980).

[May78] May, M. D., Taylor, R. J. B., and Whitby-Strevens, C., "EPL - An Experimental Programming Language," IEEE Conference on Trends and Applications: Distributed Processing, Gaithersburg, MD (May 1978), 69-71.

[May79] May, M. D., and Taylor, R. J. B., The EPL Programming Manual Distributed Computing Project, Report No. 1, Department of Computer Science, University of Warwick, Coventry, England, (1979).

[Met76] Metcalfe, R., and Boggs, D., "Ethernet: Distributed packet switching for local computer networks," Comm. ACM, 19,7 (July 1976), 395-404.

[Sho79] Shoch, J. and Hupp, J., "Performance of an Ethernet local network -- A preliminary report," Proc. Local Area Communications Network Symposium, N. B. S. and The MITRE Corp, Boston, (May 1979).

[Sho80] Shore, J. E., "The Lazy Repairman and Other Models: Performance Collapse due to Overhead in Simple, Single-server Queuing Systems," Proc. of the 7th IFIP International Symposium on Computer Performance Modelling, Measurement and Evaluation, Toronto, Canada, (May 1980), 217-224.

[Sou80] Souza, R. J., "On the Resource Requirements of Distributed Software," Technical Report, Department of Electrical Engineering and Computer Science, University of Connecticut, (Oct 1980).

[Tob79] Tobagi, F. A. and Hunt, V. B., "Performance Analysis of Carrier Sense Multiple Access with Collision Detection," Technical Report 173, Computer Systems Laboratory, Stanford University, Stanford CA, (June 1979).

Part VII

The Impact of Hardware Interconnection Structures on the Performance of Decentralized Software

by

R. Souza and E. Balkovich

Abstract

The results of an investigation of the relationship between software structure, hardware interconnect structure and the performance of decentralized computer systems are presented in this paper. Programs written in a language which has the salient features of most languages suggested for decentralized software were analyzed using trace-driven simulation. Among the results is the fact that reasonable performance may be obtained at relatively low bandwidths using typical decentralized interconnect structures. Additional results provide some insight into the use of hardware interconnect structures by decentralized software.

I Introduction

While a number of distributed systems have been constructed and described in the literature [Jen78], a design methodology for distributed systems does not yet exist. Such a methodology will certainly be based on models of the distributed software and hardware architectures. Such models will require information about the requirements that a distributed software architecture places on the communications and commputational resources of the system.

Programming language research has produced a number of suggestions for languages for distrubuted commputing [BrH78] [Hoa78] [Coo80]. A central characteristic of these languages is the concept of using a number of cooperating autonomous processes to acomplish a task. These processes execute concurrently, share no variables, and communicate with one another by passing messages. Additionally, the processes tend to be small and communi-

cate frequently. Thus, there is potential for parallelism in the software architecture, but the delays associated with message passing will certainly be an important factor in evaluating system performance.

The communications subsystem characteristics and the interface between the communications subsystem and the operating system kernel will have a large impact on system performance. Due to cost, reliability and extensibility requirements, only a few of the many hardware interconnection architectures are suitable for use in a distributed computing system. This raises the question of how the software architecture should be designed in order to best utilize the hardware.

This paper presents the results of an investigation conducted in order to provide some insight into the use of computational and communications resource by applications programs written in the distributed programming language EPL [May79]. The performance of two representative algorithms written in EPL was measured using trace-driven simulation [Fer78] in order to quantify some of the performance tradeoffs made in the design of a distributed computing system. An understanding of these tradeoffs will allow for a more methodical design of distributed systems. Of particular interest in this research are the performance characteristics of a decentralized software architecture running on a system constructed from a number of processors interconnected by an Ethernet type [Met78] contention bus. Such a system is available for experimentation. The expected performance of such a system will be evaluated and compared to other better-understood architectures, such as those using shared memory to communicate, with the intent of assessing the impact of the hardware interconnect structures on the performance of the distributed system.

II Background

EPL

The programming language EPL is representative of a number of languages suggested for programming decentralized computer systems. Processes (called acts) may be created dynamically. An instance of a process is an actor , and all actors execute concurrently. Multiple instances of the same act may be created. Actors may communicate only by passing messages, so variables may not be shared among actors. EPL programs are independent of the hardware interconnect structure of a decentralized system. As long as the appropriate operating system kernel is used, the same EPL program may be executed on a variety of decentralized architectures or a single CPU.

In addition to communicating information between actors, messages may also be used to synchronize actors. Messages between actors are not buffered. Therefore a sending actor is delayed until the receiving actor is ready to receive. Similarly, a receiving actor is delayed until the sending actor is ready.

A Decentralized Kernel

In order for a computer system to be decentralized, all software, including the operating system, must be decentralized. Operating systems for multicomputer systems which are not decentralized may maintain global state information in a central location, perhaps a table in shared memory. The kernel for a distributed operating system must distribute this information

among the processors in the system. This distribution may be performed in a number of ways. For example, each kernel may maintain its own copy of the global state information (referred to as a distributed kernel). A kernel which performs an operation which changes the global state information must make sure that all other processors receive the new state information. This problem is analagous to maintaining the consistency of a distributed data-base. A number of algorithms have been proposed for performing the data base update in a decentralized fashion.[Dij74] Alternatively, each kernel may maintain information only about only those processes on the local processor. If information about a non-local process is required (e.g. to send or receive information to or from a process on another processor), the two kernels must communicate in order for the information to be obtained. This implementation will be referred to as a partitioned kernel. A partitioned kernel system has been implemented and is available for experimentation [Fon80].

Hardware Interconnect Structures

In order to construct a completely decentralized system, there must be no central hardware component[Ens78]. This requirement restricts the number of interconnect structures from which one may construct decentralized systems. Additionally, it is advantageous if the interconnect structure be inexpensive, reliable and readily extensible. The Anderson-Jensen taxonomy of multicomputer systems is a reasonably complete enumeration of interconnection structures [And75]. Of the ten architectures in this taxonomy, only three, the loop (DDL), global bus (DSB), and fully-interconnected (DDC), do not exhibit some form of centralization. Of these three, the DDC system possesses poor extensibility and cost characteristics and is not

reasonable for use in most distributed systems.

III Analysis Approach

A reasonable way to determine the characteristics of applications written in a distributed programming language is to measure actual programs. This was the method used in this research. The measurements were performed in the following manner: A single-CPU version of an EPL runtime kernel was instrumented to provide a trace of all significant events (sends, receives, etc.) which occurred while a program was running. This trace output could then be analyzed to provide information such as the number of actors created, the names of all the actors, a histogram of message lengths and a matrix of communication traffic between actors. Since the EPL programs measured were deterministic, the information obtained in this manner is independent of the hardware interconnection architecture.

The trace information could also be used as input to a trace-driven simulation package which would simulate the execution of the programs on a variety of multicomputer architectures.

Simulation Models

This section describes the models used in the trace-driven simulation.

Kernel Model

The simulation kernel models the partitioned kernel as described above. Identical copies of the kernel are maintained in each processor. Each kernel possesses state information only for local processes. A process wishing to send to or receive from a process on another processor is

blocked while the two kernels exchange state information. Consider the case where a process wishes to send to a process on another processor. The sending process is blocked when its kernel determines that the receiving process is not resident. The sender's kernel sends an enquiry to the receiver's kernel. When the receiving process is ready to receive, its' kernel sends an acknowledgement to the sender's kernel. Upon receipt of the acknowledgement the sending process is marked as ready to run. When rescheduled, it will transmit the message text to the receiver. The receiver remains blocked until the text is received and then is marked as ready to run. A total of three message are required to pass a message between two processes on different processors: an enquiry for state information, an acknowledgement with the state information, and the text itself.

Local Actor Scheduling

A FIFO queue of actors ready to run is maintained by the kernel at each processor. The actor at the head of the queue is given control when the processor is idle. Note that an actor which has become ready to run will not immediately execute, but must wait for those queued ahead of it.

Global Actor Scheduling (Process Allocation)

The problem of process allocation is complex and we do not attempt to solve it here. It is hoped that the results of this work may be used in an optimal task allocation algorithm.[Chu80] The simulation has two algorithms for process allocation. The first of these algorithms maintains a scheduling policy which allows one to write distributed software which gracefully degrades with failure [Bal79]. The second algorithm tries to minimize com-

munications between the processors.

Logical Ring Data structures implementations in distributed programming languages are often chains or trees of processes [Tay79]. The process allocation algorithm views the processors as connected in a logical ring. When a process creates another process, the child process is placed on the logically adjacent processor. The intent is to avoid a situation in which parent and child are resident on the same processor. If the software structure is constructed appropriately, failure of a processor may disrupt parent-child communication, but not parent-grandchild communication, for example. No check as to the loading of the logically adjacent processor is performed.

Minimum Communications Cost This allocation will attempt to minimize communications between processors by creating a child process on the same processor as its' parent. The assumption, borne out by measurement, is that most process intercommunication is parent-child in nature. If the local processor is more that fifty per cent utilized, the child process will be created on the logically adjacent processor.

The Process Model

A process may be characterized by three parameters: a sequence of sends and receives, the time duration between sends and receives, and the size of the messages sent by that process. The send/receive sequence for each process is determined by processing the trace output from the single-CPU instrumented kernel and forms the input to the simulator. This processing also determines the average message length for all processes. This is the values used for message length for all messages in the system, in-

cluding kernel-kernel communication. The time between send/receives for typical EPL programs has been measured by counting instructions [Bal80]. Based on these measurements, the time between sends and receives is taken to be exponentially distributed with a mean of 10 machine instructions.

Hardware Interconnection Models

The simulation provides models for a number of different hardware interconnection systems. Several of these are described below.

Shared Memory System

This model contains from one to five processors which communicate through a shared memory. The time required to insert or remove a message in/from the memory is assumed to be zero. This configuration corresponds to the DSM architecture of Anderson and Jensen and may be used as a base from which the costs of distribution may be measured. Additionally, the degenerate case of this architecture is a uniprocessor, so that the distributed/uniprocessor tradeoffs may be measured.

The same kernel model is used in this system as the decentralized systems, but since all communication between processes is on the same processor, the kernel need not send messages to obtain state information. That is, the overhead involved in sending a message between two processes is not required to include the time required for kernels to exchange state information.

Global Arbitrated Bus (ICSB)

Processors are connected by a global bit-serial bus with FCFS arbitra-

tion. Each processor communicates with its' I/O channel via DMA, so that I/O and processing may overlap. A queue of processes wishing to access the global bus is maintained. Bus bandwidths from 10 Kbit/Sec. to 1 Mbit/Sec. were investigated. It should be noted that the arbitration method used by this interconnection architecture requires a centralized arbiter. It is not a decentralized architecture, but it was included in the analysis since it is an architecture which makes most efficient use of the available bus bandwidth. and provides a baseline from which to measure other bus interconnection methods.

Contention Bus (DSB)

This configuration models a Ethernet-type contention bus. [Met76] Interfaces listen before sending and only transmit on the bus if it is not in use. An interface can determine the presence of information on the bus (carrier detect) in zero time. This is somewhat unrealistic in a physically distributed system. If the bus is in use, the interface waits until the bus becomes free (no carrier), delays an exponentially-distributed random time, and tries to send again. Note that it is possible for the bus to be busy again at this time, causing the interface to retry again. If a collision occurs, both interfaces involved retry an exponentially-distributed random time after the bus becomes free. This model differs slightly from the Ethernet in that a collision is not detected until the end of a message transmission so that a collision destroys both messages and makes the bus unusable for a message time. Additionally, the backoff time after a collision is generated by an exponentially-distributed random number generator rather than the binary exponential backoff algorithm. The bus bandwidths investigated were the same as for the ICSB system.

Sample Application Programs

Programming in a distributed programming language requires application of a set of concepts quite different from those used when programming in a sequential language [Tay79] [Hoa78] [BrH78]. A fundamental difference is the concept of using a process to encapsulate an element of data. An array may be implemented by a string of processes, for example. The directory for a filing system may be contained in such an array, and searching the directory for an entry would involve moving down the chain of processes until the required entry is found. A process may also be used to implement a procedural abstraction, such as converting an integer to a character string. Such a procedural abstraction differs from the implementation in a sequential program, as it is concurrent with the procedures using it.

Two sample application EPL programs were selected for study. It is felt that these programs possess the characterics of a large number of programs written in languages for decentralized software. These programs compute prime numbers by the sieve of Eratosthenes and perform a heap sort on a random array of numbers.

Sieve of Eratosthenes

The sieve generates prime numbers by creating a pipeline of actors driven by a number generator. Each actor in the pipeline encapsulates a prime number and sieves out all numbers passed to it which are multiples of that prime number. A new actor is created to encapsulate a number which cannot be divided by any actor in the pipeline. The sieve is an example of

the concept of encapsulating data within a process.

Heap Sort

The heap sort generates an array of random numbers using a process to implement the random number generator and then sorts the numbers using a heap sort algorithm. Each node in the sort tree is a process. The processes are created as the tree is loaded and deleted as the tree is unloaded. A difference between the heap sort and the sieve is that the sieve tends to create an ever-lengthening chain of processes, while the heap sort creates a large number of processes arranged in a tree and then removes them. The communications requirements of the heap sort are felt to be quite different than those of the sieve, which will allow for evaluation of two extremes of this spectrum.

IV Results

Figures 1 and 2 illustrate the effects of the different interconnection structures and scheduling algorithms for the sieve and sort programs. Several items are worthy of note. The first is that there is little difference in performance between systems communicating via a high speed global bus (1Mbit/Sec) and those communicating via shared memory. While messages between processes are sent frequently, the average message size is small (around 2 bytes) and this places little demand on the interconnection system.

The logical ring allocation algorithm offers better performance than the minimum coummunications allocator for the bus architectures. This is because communications resource is so abundant that the loss in potential

parallelism caused by placing child processes on the same processor as the parent outweighs the communications delay.

Addition of a second processor with the logical ring scheduler dramatically impacts the completion times of the two programs. The sieve algorithm has two source of parallelism: the process which generates integers and feeds the pipeline; and the first sieve in the chain. Processes further and further down the chain are activated less and less frequently. Addition of the second processor with the ring scheduler places these two processes on different processors, allowing them to run concurrently. Similarly, the root process of the heap sort can run nearly all the time. Addition of the second processor allows this to happen, and the additon of further processors contributes equitably to the performance.

The fact that a 1 Mbit/Sec global bus can provide performance equal to that of a shared memory system for this class of decentralized system raises the question of how bus bandwidth in a bus-interconnected system effects performance. Indeed, the bus bandwidth will surely be a large factor in determining system cost.

Figures 3 through 6 present results of a set of experiments at various bus bandwidths below 1 Kbit/Sec. Results are presented here only for the sieve algorithm, but results for the sort are similar. The performance of a 100 Kbit/Sec bus was roughly the same as a 1 Mbit/Sec bus. Note that the information for both the ICSB (arbitrated) and DSB (contention) buses are shown to allow for comparison.

Figure 3 plots completion time as a function of bus bandwidth. Performance is dramatically increased at bus bandwidths above about 25 - 50

Kbit/Sec. This is the point at which the system changes from an I/O bound mode to a processor-bound mode of operation. This is borne out be Figure 4, which shows very high bus utilization below about 50 Kbit/Sec, and by figures not presented here which show a sharp drop in the size of the communications controller queue size as bus bandwidth approaches 50 Kbit/Sec.

Bus utilization increases as more processors are added to the system. The arbitrated system achieves a higher bus utilization, especially at low bandwidths. The contention system behavior parallels that of the arbitrated system because the interfaces can detect carrier in zero time. The model is currently being modified to associate a time delay with carrier detection. The performance degradation with the contention bus is probably due to the fact that the random delays generated by the controller may place messages on the bus out of order. This second order effect requires further investigation.

Figures 5 and 6 show the effects of bus bandwidth on processor utilization and throughput. These increase somewhat linearly with bandwidth, indicating that there is a backlog of work at each processor.

Summary

Trace-driven simulation was used to measure the anticipated performance of two representative programs written in the distributed programming language EPL. The behavior of these programs on a number of multicomputer systems was measured.

Performance of these programs was effected by a number of factors, especially the bandwidth of the intercommunications medium. System perfor-

mance is essentially CPU-bound until communications bandwidths are quite low, at which point performance degrades significantly as the system becomes I/O bound. Another contributing factor is the process allocation algorithm. Two simple algorithms resulted in quite different system performance characteristics. Optimal performance will require application-dependent process allocation and selection of an appropriate interconnection subsystem bandwidth.

References

[And75] Anderson,G.A, and Jensen,E.D, Computer Interconnection Structures: Taxonomy, Characteristics and Examples, ACM Computing Surveys, Vol. 7, No. 4, Dec. 1975.

[Bal79] Balkovich, E. E, A Structure for Decentralized Software, Technical Report CS-79-9, Department of Electrical Engineering and Computer Science, University of Connecticut

[Bal80] Balkovich, E. E., and Whitby-Strevens, C., On the Performance of Decentralized Software, Proc. of the 7th International Symposium on Computer Performance Modelling, Measurement and Evaluation, Toronto, Canada, May 1980, p 173.

[BrH78] Brinch-Hansen, P., Distributed Processes: A Concurrent Programming Concept, CACM, Vol. 21 No 11, November 1978, p. 943.

[Chu80] Chu, W.W, et. al., Task Allocation in Distributed Data Processing, Computer, Vol. 13 No. 11, November 1980, p. 57.

[Coo80] Cook, R.P., *MOD - A Language for Distributed Programming, IEEE Trans. on Software Engineering, Vol. 6, No. 6, Nov 1980, p. 57.

[Dij74] Dijkstra, E. W., Self-Stabilizing Algorithms in Spite of Distributed Contr... CACM, Vol 17., No. 11, November 1974, p 643.

[Ens78] Enslow P., What Is A "Distributed" Data Processing System?, Computer, Vol, 11 No 1, Jan. 1978 p 13.

[Fer78] Ferrari, D. Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Fon80] Fontaine, S., A Distributed Computer Operating System Kernel, M.S. Thesis, Department of Electrical Engineering and Computer Science, December 1980.

[Hoa78] Hoare, C.A.R., Communicating Sequential Processes, CACM, Vol. 21 No. 8, August 1978, p 666.

[Jen78] Jensen, E. D., The Honeywell Experimental Distributed Processor - An Overview, Computer, January 1978, p. 28.

[May79] May, M.D and Taylor, R.J.B, The EPL Programming Manual, Department of Computer Science, University of Warwick, May 1979.

[Met76] Metcalfe, R.M, and Boggs, D.R., Ethernet: Distributed Packet Switching for Local Computer Networks, CACM, Vol. 19, No. 7 July 1976, p 395.

[Tay79] Taylor, R.J.B., Notes on Programming in EPL, Department of Computer Science, University of Warwick, May 1979.

FIG 1: SIEVE COMPLETION TIMES

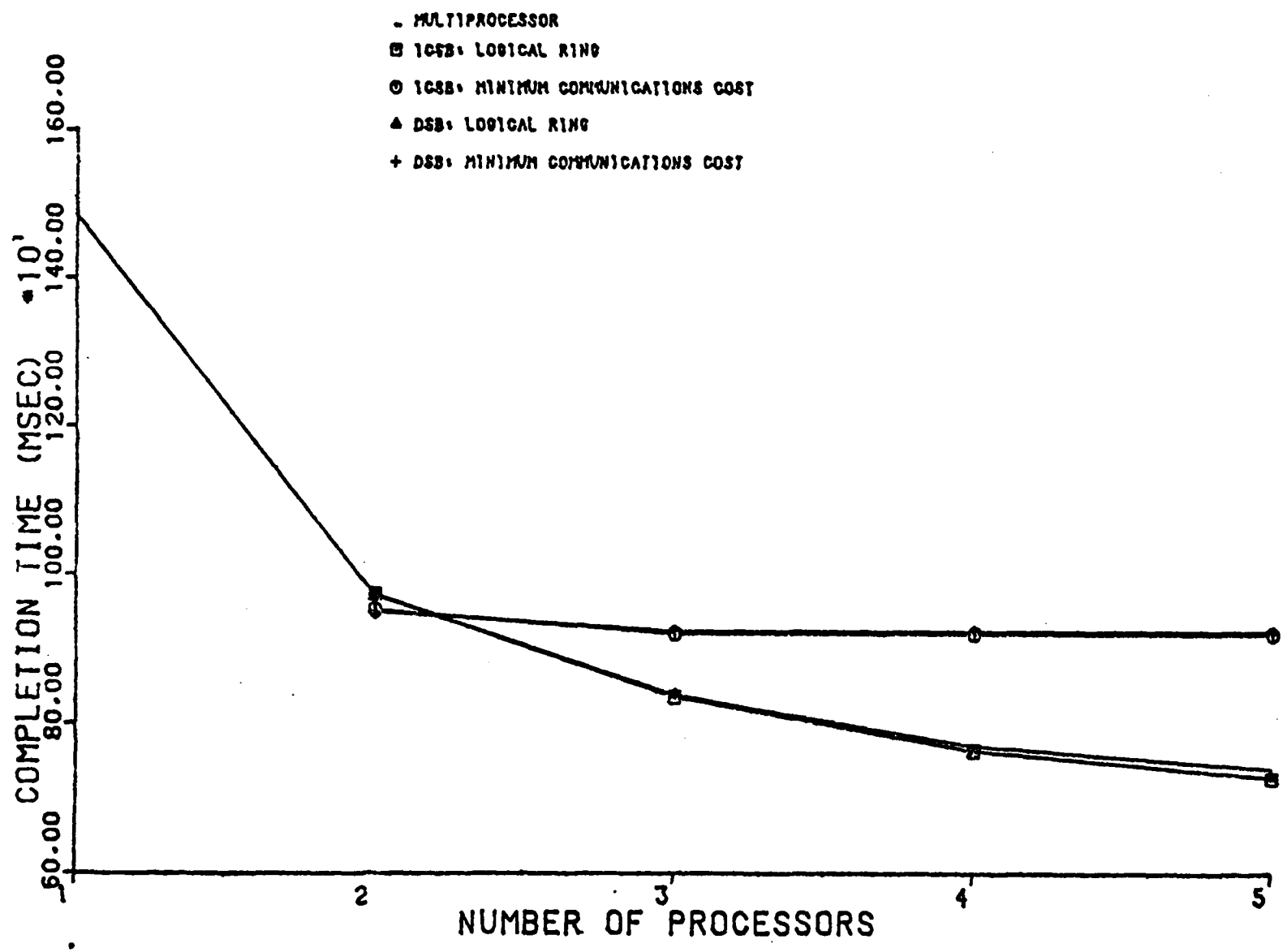


FIGURE 2: SORT COMPLETION TIMES

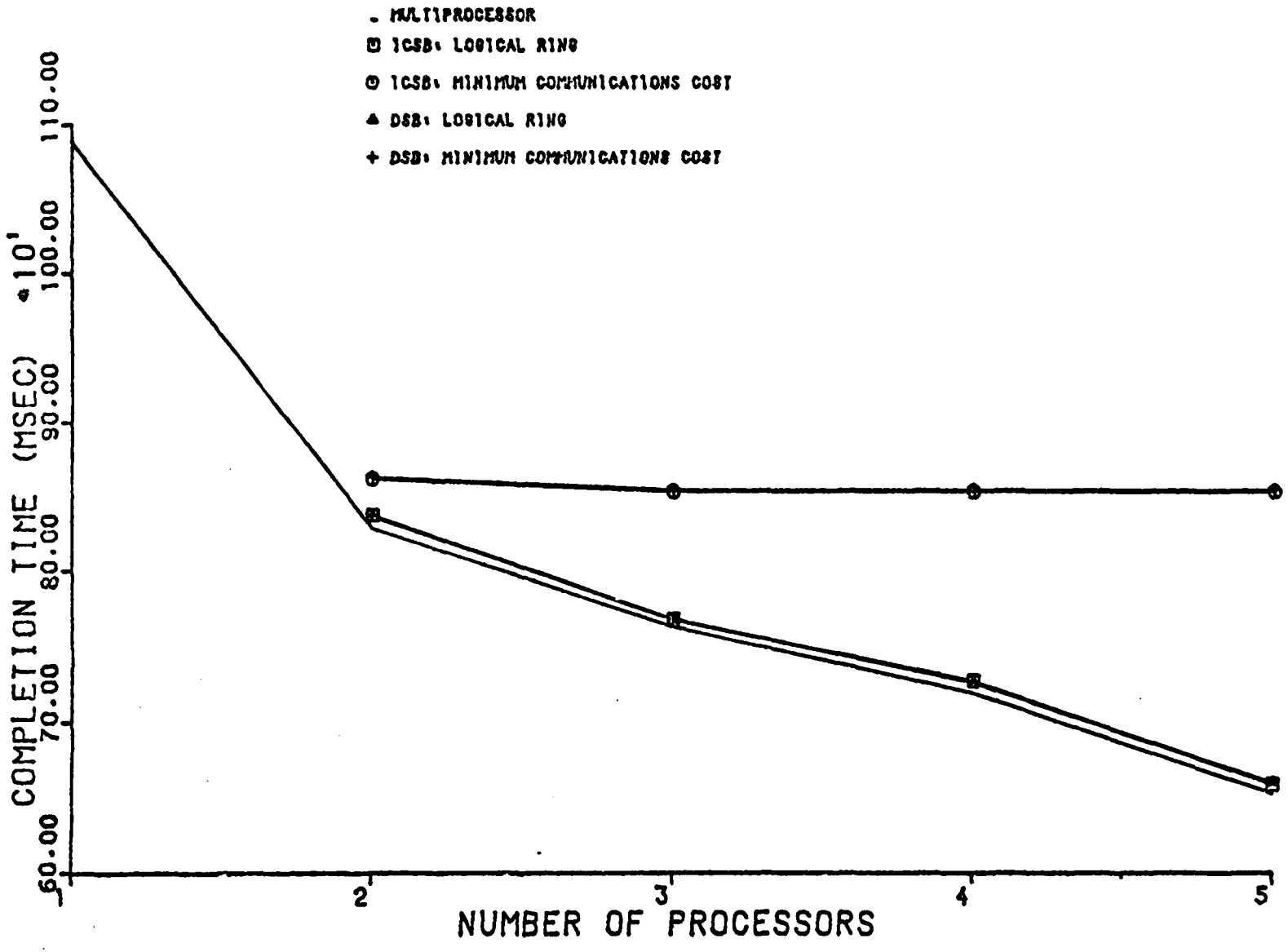


FIG 3: COMPLETION TIME VS BANDWIDTH

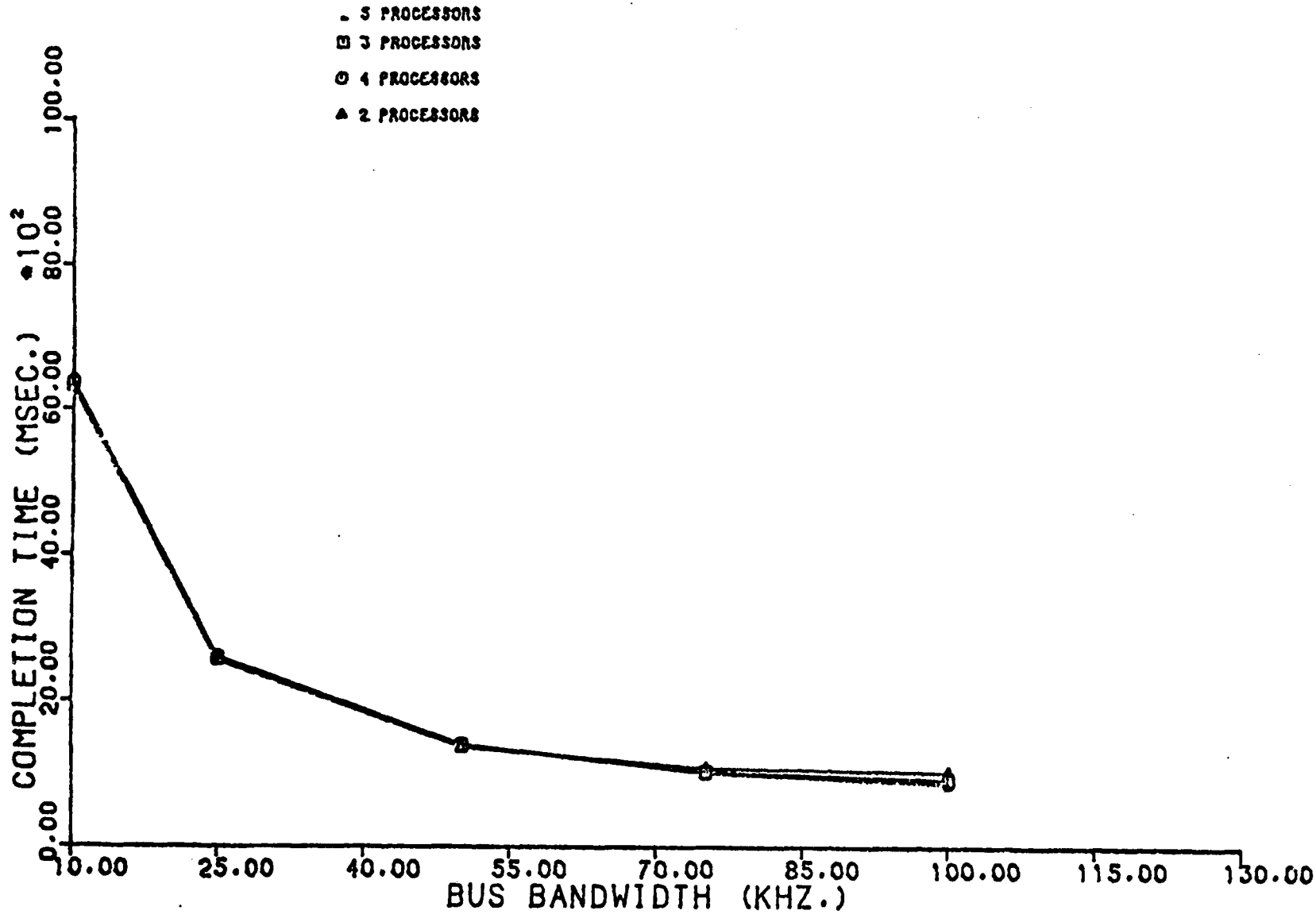


FIG 4 BUS UTILIZATION VS BANDWIDTH

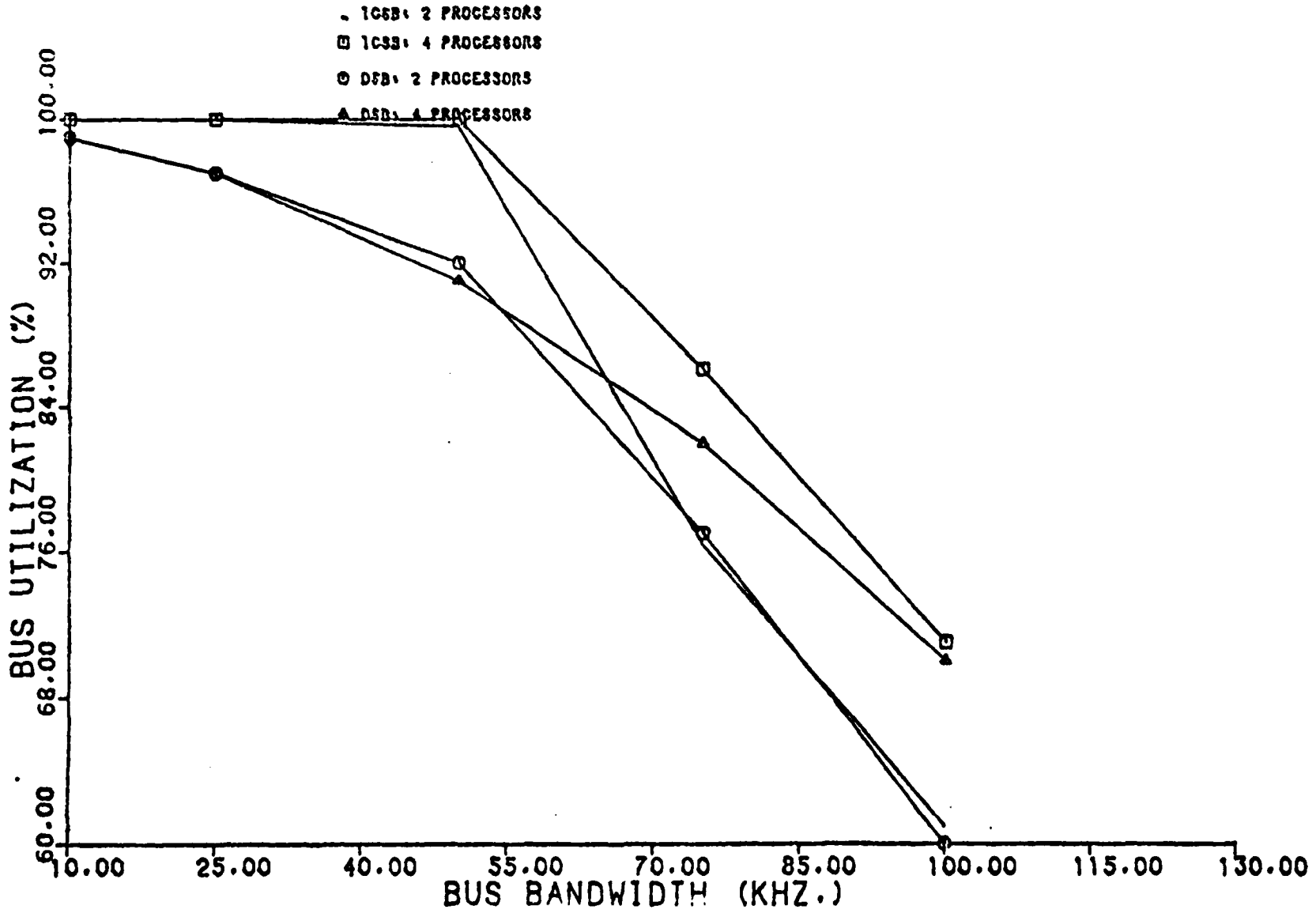


FIG 5: PROCESSOR UTILIZATION VS BANDWIDTH

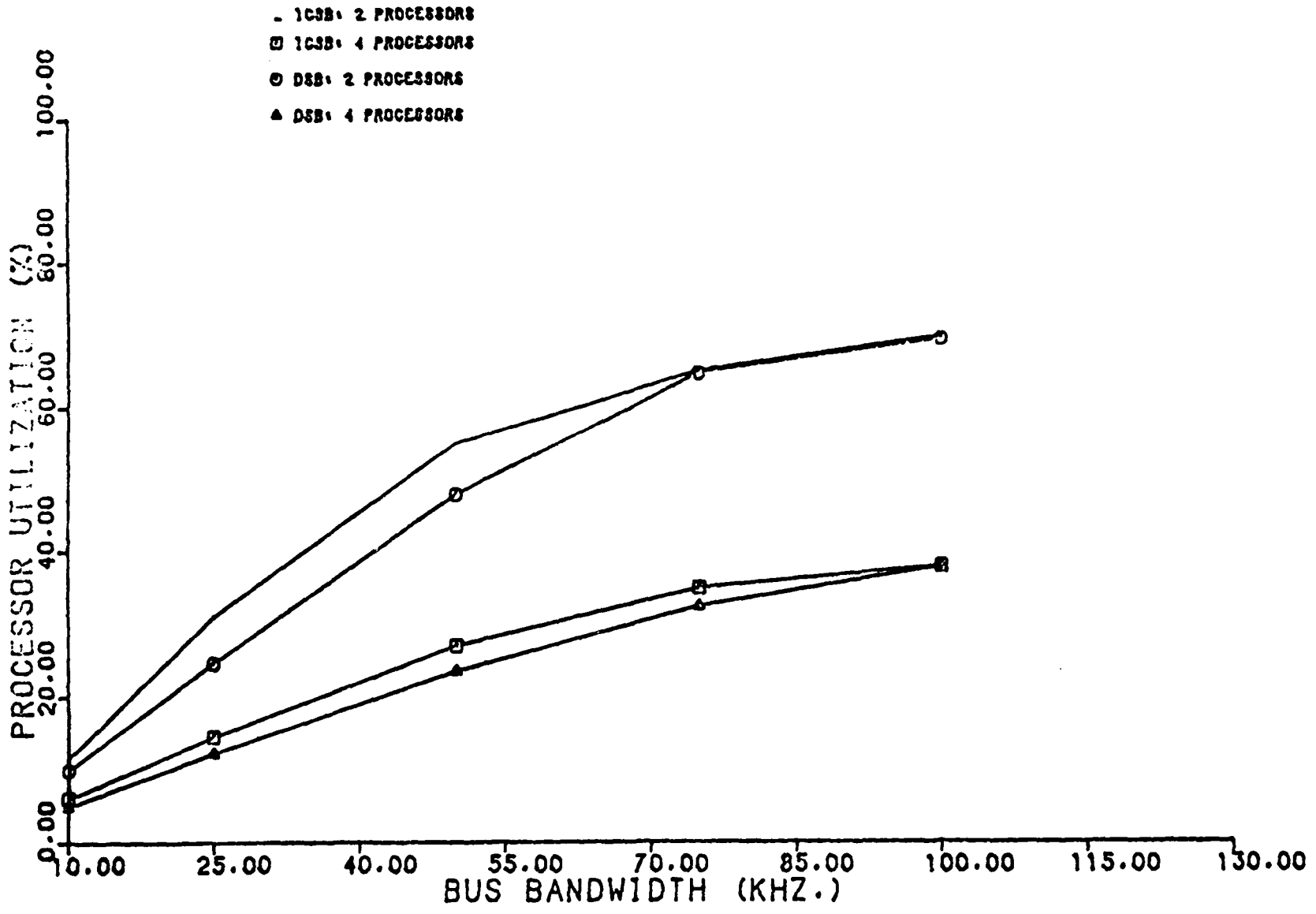
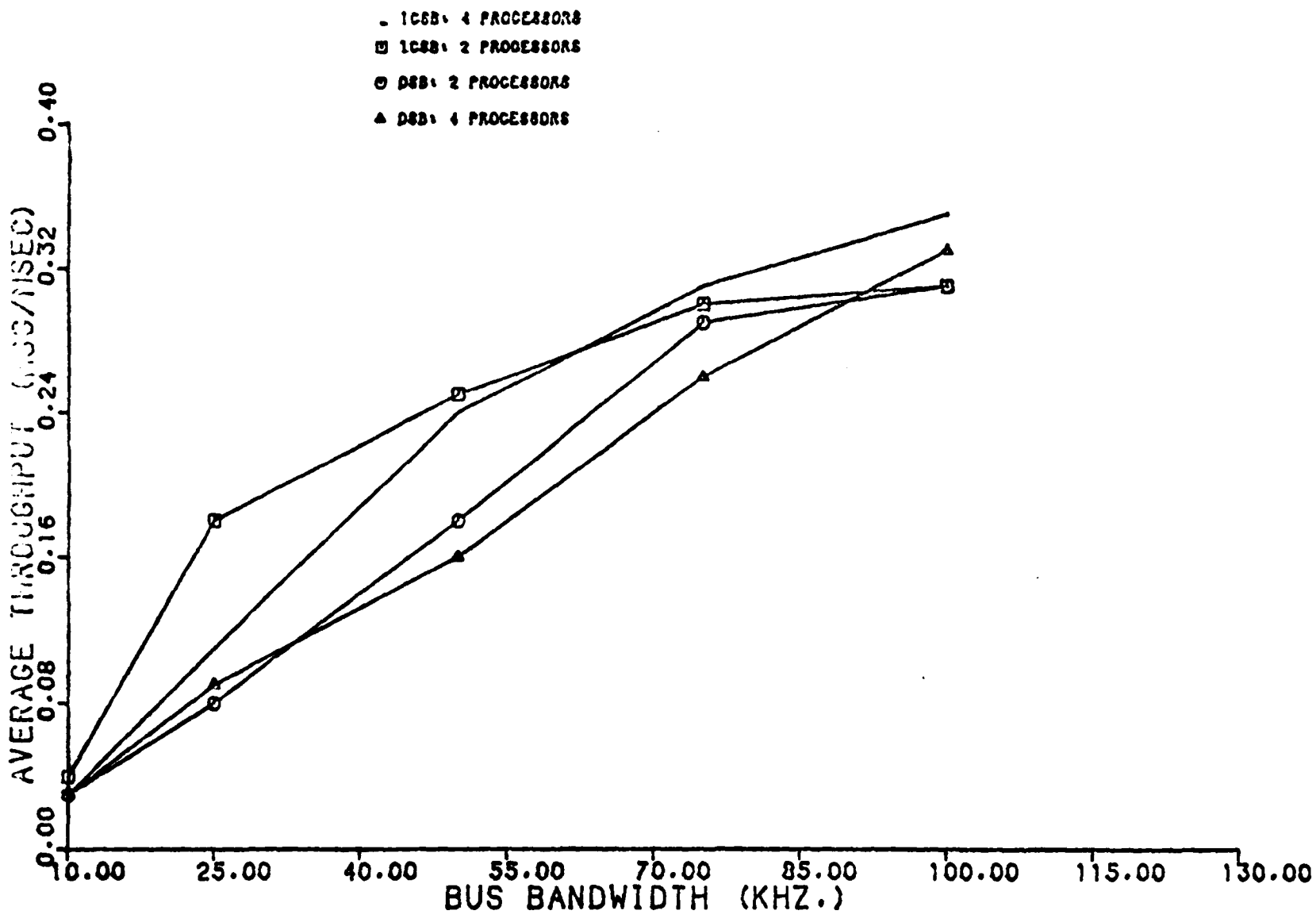
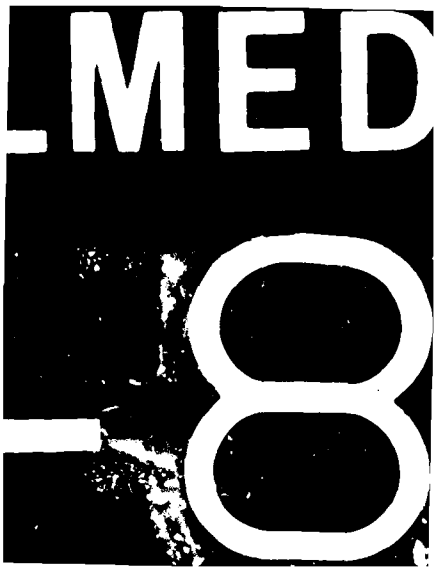


FIG 6: THROUGHPUT VS BANDWIDTH




LMED
8