CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

The Homogeneous Multiprocessor: A Simulation Study and an Operating System Design

Kin Fun Li

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy at
Concordia University
Montréal, Québec, Canada

July 1987

ISBN 0-315-37109-9

ABSTRACT

The Homogeneous Multiprocessor:
A Simulation Study and an Operating System Design

Kin Fun Li, Ph.D.
Concordia University, 1987

The Homogeneous Multiprocessor is a tightly-coupled, MIMD architecture consisting of a number of identical processing elements, and two types of interconnection pathways: (1) a distributively controlled network of switches ("extended buses") which permits each processor to access the memory of its two immediate neighbours, and (2) a fast local area network (H-Network) which permits each processor to communicate with all the other processors in a point-to-point or broadcast mode. Experiments to study the behaviour of the switching network under a varying interprocessor communication demand, and to obtain a performance measure of the Homogeneous Multiprocessor by simulating distributed algorithms, are carried out on a simulator written for the Homogeneous Multiprocessor. The operating system is based on a distributed operating system nucleus — the HM-Nucleus, which has a hierarchical design that provides a distributed test-bed environment for distributed application programming, in which the prominent features of the hardware structure are presented to the user in easy-to-use abstractions. Operating system support for distributed applications on the Homogeneous Multiprocessor architecture includes interprocessor communication mechanisms, distributed data structure management, and global naming mechanisms.

ACKNOWLEDGEMENTS

I wish to express my sincerest gratitude to my thesis supervisor, Dr. N. J. Dimopoulos, for his caring guidance throughout my undergraduate and graduate studies. He was always available to direct and to advise whenever the occasion arose.

I would like to thank all the people who have been associated with this project, especially Marc Alain and James Demers for their assistance in software coding, and T. Segal and M. Robillard for making the hardware operative.

I would also like to acknowledge Centre de recherche informatique de Montréal, inc., for its bursary award for the period from September 1985 to August 1987, and the support from the National Sciences and Engineering Research Council of Canada, and Fonds F.C.A.R. pour la formation de chercheurs et l'aide à la recherche.

Finally, I like to thank Dr. J. W. Atwood for his valuable comments. These helpful suggestions could have only been provided by someone with much experience, knowledge, and patience.

Dedicated to my Parents,
Leung and ShuiYing

and to my Aunt,
LanSin

Table of Contents

List of Figures

List of Tables

List of Appendices

1. Introduction

Advanced development in the past decade in microprocessor hardware and electronic circuit fabrication, and the desire to have larger, faster computing machines, have prompted increased research activities in the area of computer architecture. Either using off-the-shelf, high-performance, commercially available products, or implementing complex electronic circuits on IC-chips using Very Large Scale Integration (VLSI) technology, computer designers have proposed numerous architectures for distributed and multiprocessing systems. These systems have great potential in improving the processing power, capacity, speed, and resiliency of our present-day systems. This fact is readily apparent when such systems are compared with conventional uniprocessor systems. In addition, the software and hardware modularity nature of distributed and multiprocessing systems makes the cost-performance factor attractive. A wide range of multiprocessing systems has been proposed and built with varying degrees of coupling and homogeneity. C.mmp [69], Cm* [64], NYU Ultracomputer [29], Butterfly [14], and Cosmic Cube [59] are but a few examples

Formally defined, a multiprocessing system is a computing system composed of two or more processors that are capable of independent instruction execution and information exchange through some interconnection mechanism. In existing systems, interprocessor communication pathways vary from the cross-bar switching in the C.mmp and the Omega-network in the NYU Ultracomputer, both using shared memory, to the message passing network found in the Cosmic Cube. There are also systems that implement both communication mediums, for example, the Kmap in Cm* offer both networking and shared memory facilities to the users.

The C.mmp developed at Carnegie-Mellon University is a multiprocessor computer system using minicomputer processors [69]. The non-homogeneous system uses modified minicomputers (PDP-11/20 and PDP-11/40) as processing elements. The primary

memory available is 2.7M bytes. Sixteen processors, and primary memory arranged in sixteen ports are connected by a 16X16 cross-point switch. Each processor is connected to all the memories through a relocation unit, which relocates the 18-bit processor-generated Unibus address into a 25-bit address space that maps into one of the memory ports. Memory contention is resolved by the cross-point switch using a hardware priority-ordered request queue.

The Cm* project at Carnegie-Mellon is the implementation of a multiprocessor with shared memory organised in a tree hierarchy [64]. A unit of memory and a processor are closely coupled in each Computer Module (Cm) and access to non-local memory is made via a network of buses based on packet switching. The hardware of individual Computer Modules consists of a DEC LSI-11 processor, an Slocal (local switch), and standard LSI-11 bus memory and devices. Up to fourteen Computer Modules and one Kmap form a cluster. There are three levels of memory access: (i) local — within a Computer Module through the local bus; (ii) intracluster — within a cluster of Computer Modules through the Map Bus; and (iii) intercluster — among clusters through the Intercluster Bus. The address mapping elements are the Slocals within the Computer Module and the Kmaps that direct intercluster and intracluster memory references. Any processor can access any memory location in the system. The routing of a processor's reference to a target memory is performed by the Slocals and the Kmaps, and is transparent to the users.

The NYU Ultracomputer project at New York University is the proposal of a shared-memory parallel machine composed of autonomous processing elements [29]. N processing elements are connected to N memory modules through a message-switching network with the geometry of an Omega-network. Each processing element, which is custom-made, is attached to the network via a processor network interface, and each memory module is attached to the network via a memory network interface. Interprocessor synchronisation is achieved through the fetch-and-add primitive. This primitive is

implemented in hardware within the memory network interface in the enhanced Omega-network, where a queue is associated with each switch to enable concurrent processing of requests for the same memory port.

The Butterfly Parallel Processor is a homogeneous, tightly coupled, shared memory multiprocessor, commercially available from the Bolt, Beranek and Newman Inc. [14]. A Butterfly system can be configured with from 1 to 256 processor nodes. Each node consists of a MC68000 processor, 1 MByte of memory (expandable to 4 MByte), a co-processor called Processor Node Controller, an I/O bus interface to Multibus, and an interface to the Butterfly switch. The topology of the switch is similar to the Fast Fourier Transform Butterfly. Each Butterfly switching node is a custom VLSI chip consisting of a 4 input—4 output switching element. Eight VLSI switch chips thus form a 16 input—16 output Butterfly switch (which is expandable to accommodate more processors). There is a path through the switching network from each node in the system to any other node. Therefore, each processor can access any memory location in the machine, using the Butterfly switch to make remote references The Processor Node Controller is responsible for transmitting and receiving messages over the switch, and for the communication and synchronisation of data access among nodes. In addition to single word (16-bit) transfers, it can also transfer block data up to 32 Mbit/sec over the switch.

The Cosmic Cube at Caltech [59] is a homogeneous multiprocessing system consisting of 64 small computers connected by a network configured in a six-dimensional hypercube fashion. Each node has an Intel 8086 as instruction processor, an Intel 8087 floating-point co-processor, and has a memory size of 128K bytes. Each of these nodes is connected through bidirectional, asynchronous, point-to-point communication channels to six other nodes. A distributive application is expressed in terms of a collection of communicating (message passing) concurrent processes residing in the same or different nodes. The Cosmic Cube is a hardware simulation of a future VLSI implementation

that is expected to consist of single-chip nodes, so are the other hypercube structures — Mark II & III at Jet Propulsion Laboratory, Connection Machine of the Thinking Machines Corp., and NCube/ten of the NCube [67].

1.1. The Homogeneous Multiprocessor Architecture

One such proposal for a multiprocessing system is the Homogeneous Multiprocessor [22], a microprocessor-based, tightly-coupled, multiple-instruction/multiple-data (MIMD), machine featuring both networking and shared memory facilities The individual processors are the Motorola MC68000 microprocessors incorporated with the Motorola MC68451 Memory Mangement Unit. Intended applications include distributed simulation, relaxation processing, image processing, and speech processing.

Figure 1.1 displays the tightly coupled architecture of the Homogeneous Multiprocessor. A typical system is composed of k ($k \geq 3$) processing elements, k memory modules, k+1 interbus switches s_i that isolate the processing elements from each other, and the H-Network, which is a fast local area network used for point-to-point and broadcast-mode communications.

Each processing element P_i owns its local memory module M_i and accesses it via its local bus b_i. Each P_i also has the exclusive use of the respective network station HS_i. Interbus switches s_i separate adjacent local buses. These switches provide each processor with the ability to access the memory of either one of its two immediate neighbours. The access is facilitated through the creation of an extended bus.

An extended bus is the dynamic fusion of two neighbouring local buses effected through the closing of the intervening switch after a request from either or both processors adjacent to the switch. Once an extended bus is created, it exists for the duration of the request—normally one memory access cycle—and deteriorates to its component local buses once the request ceases to exist. The extended buses can be used by the pro-

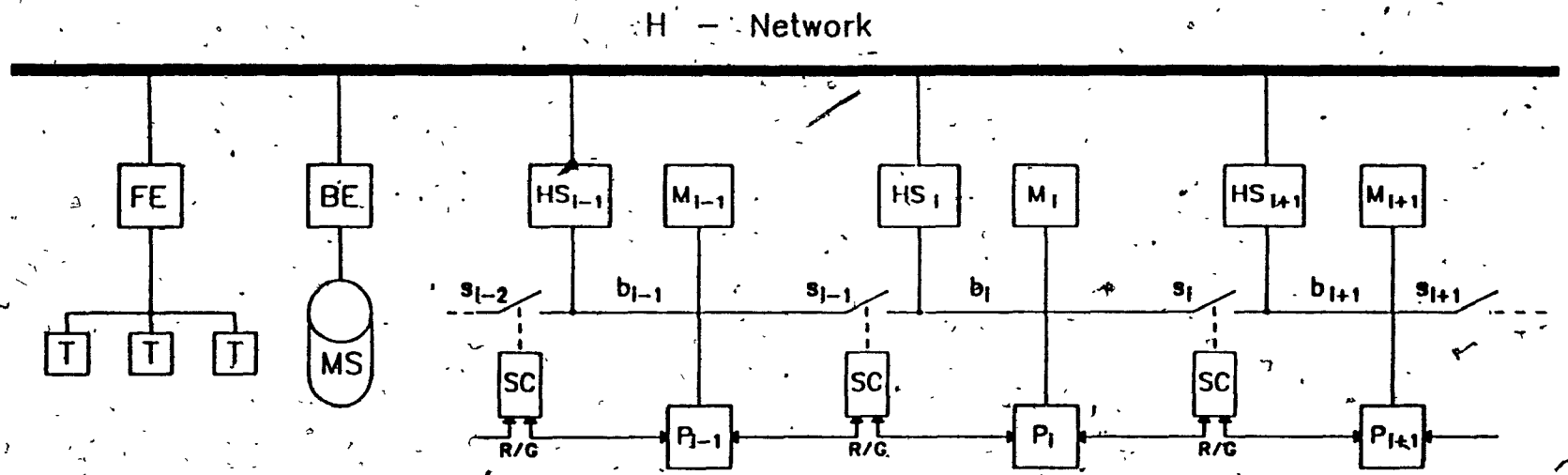


Figure 1.1 The Homogeneous Multiprocessor Architecture

P: Processor M: Memory Module s: Bus Switch
FE: Front End BE: Back End SC: Switch Controller
b: Local Bus T: Terminal MS: Mass Storage
HS: H-Network Station R/G: Bus Request/Grant

cessors of the Homogeneous Multiprocessor to communicate with either of their two immediate neighbours.

The process of creating an extended bus passes through two phases. The first phase is processor independent, and determines whether a switch can be closed through the execution of a switching algorithm, shown in Figure 1.2 [22]. The second phase is dependent on the processor, it is during this phase that a switch actually closes and forms a pathway between two adjacent processors. The decision to close a switch is taken during the first phase. Once the decision has been taken, the switch will actually close after a finite delay.

In discussing the switching algorithm, we will identify the state of a switch with the state of the algorithm without loss of generality. A switch can exist in one of the three logical states. The next state transition is based on the presence of a request for it to close and the present states of its two neighbouring switches. The three states at which a switch can exist, are:

OPEN - This state signifies that no requests exist, or, if a request exists, it will not be honoured immediately, because a neighbouring switch is currently servicing a request.

GRAY - This state signifies that a request is acknowledged and that service (i.e., switch closure) will be granted in the immediate future.

CLOSED - This state signifies that it is safe for a switch to close. The actual closure of the switch will take place during the second phase, which commences immediately after a switch enters the Closed state.

When a processor requests the access of a neighbouring memory module, it makes a request to a switch. This request is intercepted by the appropriate switch and forwarded to the memory module after the switch closes. Then the memory module acknowledges the transfer of data to the requesting processor, and this terminates the cycle. A request

For a switch S_i

If no request exists, it becomes *Open*;

Otherwise, if a request exists then:

If *Open*, it becomes *Gray*, provided that the switch to its left, S_{i-1}, is *Open*.

Otherwise, it remains *Open*.

If *Gray*, it becomes *Closed*, provided that the switch to its right, S_{i+1}, is *Open*.

Otherwise it remains *Gray*.

If *Closed*, it remains *Closed*.

The leftmost S_0, and rightmost S_k, switches are always *Open*.

Figure 1.2 The Switching Algorithm

from a processor to a switch remains asserted during the request period, which ends with an acknowledgement from the requested memory module to the requesting processor. The switching algorithm guarantees the safe (i.e., no two adjacent switches will close at the same time) and live (i.e., a switch requested to close will eventually do so) operation of the network of the interbus switches [22]

The second component of the architecture, the H-Network [18], is a high-speed (about 7M-bytes/s) local area network with a structure resembling that of the Ethernet [46]. However, it utilises separate pathways for data transmission and network acquisition. The H-Network is designed for network spans of the order of 10 metres. This fact makes the signal propagation delay extremely short, which together with the parallelism employed in its bus structure, increases the performance of the network.

The Homogeneous Multiprocessor is architecturally innovative in its ability to behave both as a distributed system coupled through a fast local-area network and as a tightly coupled pipeline of processors The H-Network provides an ideal environment for applications such as distributed file systems, distributed databases, distributed simulation, and so on. In addition, the extended bus mechanism permits the efficient implementation of such parallel algorithms as the ones found in vision and digital signal processing, and relaxation processing.

1.2. Performance Study

It is imperative to carry out a performance study of the proposed system before hardware implementation is finalised. The performance study evaluates the efficiency of the architecture, reviews any major flaws in the design, and, in the process, helps to make certain design decisions. This is especially important in the implementation of large-scale systems where savings in time, effort, and money are appreciated.

The performance study of a system (computer or not) can be achieved in general by two methods: analytical and simulation.. The analytical method is based on

mathematical models describing the system Modeling tools, such as Petri Nets [45] and queueing system models [62], are used to describe and analyse the system. Simulation, on the other hand, emulates the system states over a period of time by software [12,61]. Comparing the two methods, simulation tends to approximate the actual operation more accurately than analytical methods, since certain aspects of the system behaviour are quite often omitted to make the mathematics more tractable in an analytical model [28].

We have carried out a performance analysis on the structure of the Homogeneous Multiprocessor while we were building up the hardware. The analytical evaluation, and the simulation using PAWS (Performance Analyst's Workbench System) [9] for the H-Network can be found in [21,68], while the performance analysis for the processor array is presented in Chapter 2 of this thesis.

Our objectives in evaluating the processor array were to study (i) the behaviour of the switching network under different load conditions, and (ii) the efficiency of the Homogeneous Multiprocessor architecture in running application programs. The behaviour of the switching network is analysed because the major bottleneck in most computing system is the pathways (the extended buses in the Homogeneous Multiprocessor), that lead to the shared memory module, especially when there is heavy contention to the memory among competing processors. We also wanted to examine the Homogeneous Multiprocessor architecture for its suitability in running application programs Simulating distributed algorithms gives us an indication whether the Homogeneous Multiprocessor provides a feasible architecture for certain classes of applications

The switching algorithm that determines the connection among processors (i.e., the closure of switches) can be represented by a large sequential machine with 3^k states and k inputs, where k is the number of requests. The behaviour of such a machine is predictable. However, due to the large number of states, the analytical evaluation is infeasible. Thus, we have used simulation to study the network of switches. Simulation also permits us to monitor the actual hardware operation, which is very helpful in the

processes of designing and verifying distributed algorithms.

1.3. Issues in Software Design

A second, but equally important goal, is the design of an operating system to provide users with a programming environment, and friendly and controllable access to the raw hardware. While the hardware was under construction and verification, the operating system was designed. Code for the lower layers of a preliminary single-node operating system was tested and debugged on the first prototype node An additional benefit of this process was that it uncovered some hardware defects.

Due to its distributed hardware structure, a multiprocessor system requires an unconventional operating system that has a different conceptual and implementational approach from those of traditional uniprocessor systems. The approach to the design of an operating system for a multiprocessor system is based on the underlying hardware architecture such as the granularity of the processing elements, the degree of coupling between the processing units and memory modules, and the homogeneity of the system components.

Many of the existing operating systems designed for distributed and multiprocessing systems are based on the concept of the "kernel". For example, Hydra was implemented as the operating system kernel for the C.mmp multiprocessor system [69]. The primary goal of Hydra is to permit operating system services and facilities to be implemented as user-level (user-defined) programs on top of the kernel. Such an approach also permits the separation of policy and mechanism. Policies give user-level control (in software) over the utilisation of system resources. Mechanisms of universal applicability are provided in the kernel to implement these policies to drive the hardware. Policies are encoded in user-level software which passes parameters to the kernel where they are executed in a protected hardware environment. Other examples of kernel implementations include V Kernel [13], Roscoe [63], and Accent [54].

Another popular conceptual approach to operating system design is to provide information hiding [51]. To handle the complex operations involved, the operating system is partitioned into a hierarchy of manageable abstraction levels. This layered structure defines hierarchical levels of operations. At each level the service requested by a higher level is carried out by operations in various lower levels. The detailed operations in the sublevels are hidden from the current level and appear transparent to the invoker. Modules can be added or modified efficiently in such a layered structure. In general, such changes are pertinent to a particular level only In short, a layered approach allows modularity, information encapsulation, and functional transparency Examples of this layered approach include the THE [17] and the PSOP [48] systems

A major concern in designing an operating system for the Homogeneous Multiprocessor is to bring to the users, even the ones that are not directly connected to our project, the underlying architectural features in an easy-to-use fashion This is especially important in multiprocessing systems where the hardware configuration affects the design and implementational approach of application programs. For instance, the instruction "fetch & add" in the NYU Ultracomputer enables the users to take full advantage of the hardware's capability to access shared memory [29]. By providing these hardware abstractions, in the form of operating system support or user friendly software in the application level, we hope to create an operating environment that provides a test-bed for parallel applications programming.

Our main objective in designing the operating system is to allow system software extensibility. Any modification due to performance and functional requirements in both hardware and software should be carried out with ease and with minimal adjustments on the current system. For instance, adding processing elements to increase the performance of the Homogeneous Multiprocessor should not require a major redesign in both hardware and software. The system stays operational with the addition of the new resource being transparent. New system software installed should not affect the

operation of the existing system nor require the restructuring of existing software.

To achieve this objective, we need modularity, and information and decision encapsulation in both hardware and software. It is evident that the architecture of the Homogeneous Multiprocessor provides modularity and encapsulation in hardware. Similarly, the software design is modular. Software modularity can be achieved, by implementing basic and necessary operating system functions required to make the existing hardware a viable environment for testing and application purposes, in an encapsulated and portable module.

The operating system for the Homogeneous Multiprocessor is based on a kernel structure and follows a hierarchical or layered design. However, we use the term "nucleus" to denote the abstraction referred to as "kernel" in other systems and reserve the term kernel for the lowest layer in our model. The design of such an operating system nucleus, the HM-Nucleus, is presented in Chapter 3 of this thesis.

In a multiprocessing or distributed system, effective ways are needed for synchronisation and communication among processors. Obviously, the underlying hardware configuration of the system determines the communication techniques available to the processors for information exchange. Numerous interprocess/interprocessor communication techniques have been proposed: message passing through mailboxes [30], pipes [56], ports [32], and rendezvous [34]. For the Homogeneous Multiprocessor, two types of communications mechanisms are available: pipes that utilise the network and channels that utilise the extended buses.

It is expected that cooperating processes residing in different processors need data structures to share information in a multiprocessing environment. The shared memory among adjacent processors in the Homogeneous Multiprocessor is implemented as shared regions. The abstraction of shared region is similar to critical regions [16], monitors [31] and guarded regions [10], and it is protected by a mutual exclusion mechanism. On the global level, replicated data structures are managed by algorithms utilising both of the

available communications pathways in the Homogeneous Multiprocessor to ensure the multiple-copy consistency. However, no locking is involved in our protocols, as opposed to most existing methods [25].

The partitioning, and the logical-to-physical mapping of resources are important issues in a multiprocessing system because an inefficient naming mechanism will hinder its performance. The naming mechanism of the Homogeneous Multiprocessor is capability-based, as found often in existing systems (e.g., in Hydra [69]). Dynamic name binding is also provided by taking advantage of the broadcasting capability in the Homogeneous Multiprocessor.

Since the Homogeneous Multiprocessor provides users two communication pathways; the operating system support is designed in such a way as to bring the separate communication pathways available to the application level. Besides the general operating system facilities such as memory allocation and interrupt handling, these supporting functions include interprocessor communications, distributed virtual memory management, and global naming, and are discussed in Chapter 4.

1.4. Objectives of This Research

Summarising, the objectives of the research presented in this thesis are:

(i) To study the behaviour of the switching network under different load conditions, and the efficiency of the Homogeneous Multiprocessor architecture in running application programs, based on a simulation study.

A simulator is written for the Homogeneous Multiprocessor. Its model is described in section 2.2, while its implementation is presented in section 2.3. The simulation experiments to study the behaviour of the switching network and the efficiency of the architecture in running application programs are presented in section 2.4.

(ii) To make the Homogeneous Multiprocessor user friendly and expose its prominent architectural features in ways which permit their effective use by application programs.

An operating system nucleus is designed to provide modularity, and information and decision encapsulation. Section 3.1 discusses the overall philosophy of designing an operating system while section 3.2 examines the layered approach. The overall structure of the HM-Nucleus is presented in section 3.3. The individual layers of the hierarchy are presented in sections 3.3.1 through 3.3.8.

A layered operating system nucleus facilitates the implementation of operating system supporting functions to aid application programming. These supporting functions include interprocessor communications, distributed virtual memory management, and global naming, which are discussed in sections 4.1, 4.2, and 4.3 respectively.

In this thesis, the overall conceptual design of the operating system nucleus and its supporting functions are presented. The lowest two layers of the HM-Nucleus, namely, the Kernel and the Physical Memory Management, have been implemented. Throughout this work, implementational proposals are made for the various layers and the supporting functions, based on the conceptual design.

Using the simulator as an analytical tool for parallel applications, and the operating system nucleus and its supporting functions as implementational aids, we hope to provide a sufficient base for further research on software development and parallel application programming.

2. Performance Analysis

2.1. Current Status of the Hardware

Before giving any details on the simulator implementation and the operating system design, the following is a brief description on the current status of the Homogeneous Multiprocessor hardware.

As shown in Figure 2.1, each node in the Homogeneous Multiprocessor consists of a Motorola 8 MHz MC68000 processor, an MC68451 Memory Management Unit (MMU), 16K byte of EPROM, 32K byte of static RAM and 1M byte of dynamic RAM. The implementation of each node is based on the Motorola MC68000 Educational Board [2].

The MMU performs virtual-to-physical address mapping through the information obtained from the MMU descriptors (32 in total). The MMU will generate an interrupt to the processor if an out-of-range or protected segment is accessed. The processor can operate in either the supervisor or the user state. This distinction of operating states, together with the MMU protection mechanism, allows the implementation of a low-level protection scheme in the system.

Using memory mapped I/O addressing techniques, several peripherals are interfaced to the processor. Notable are the two MC6850 Asynchronous Communications Interface Adaptors (ACIAs) interfaced to RS-232 serial communications links, and the MC6840 Programmable Timer Module used to generate different clock rates for timing purposes.

The 16KByte EPROM houses the Motorola Tutor monitor [2], and we are using this monitor's facility to debug and trace application programs. The 1MByte dynamic RAM is interfaced to the processor and the MMU through a refreshing circuit, and an error detecting circuit. The error detecting circuit will generate an interrupt to the processor if an error is detected in the dynamic RAM. This 1MByte memory (expandable to 4MByte) is directly accessible by the local processor and is also available to the

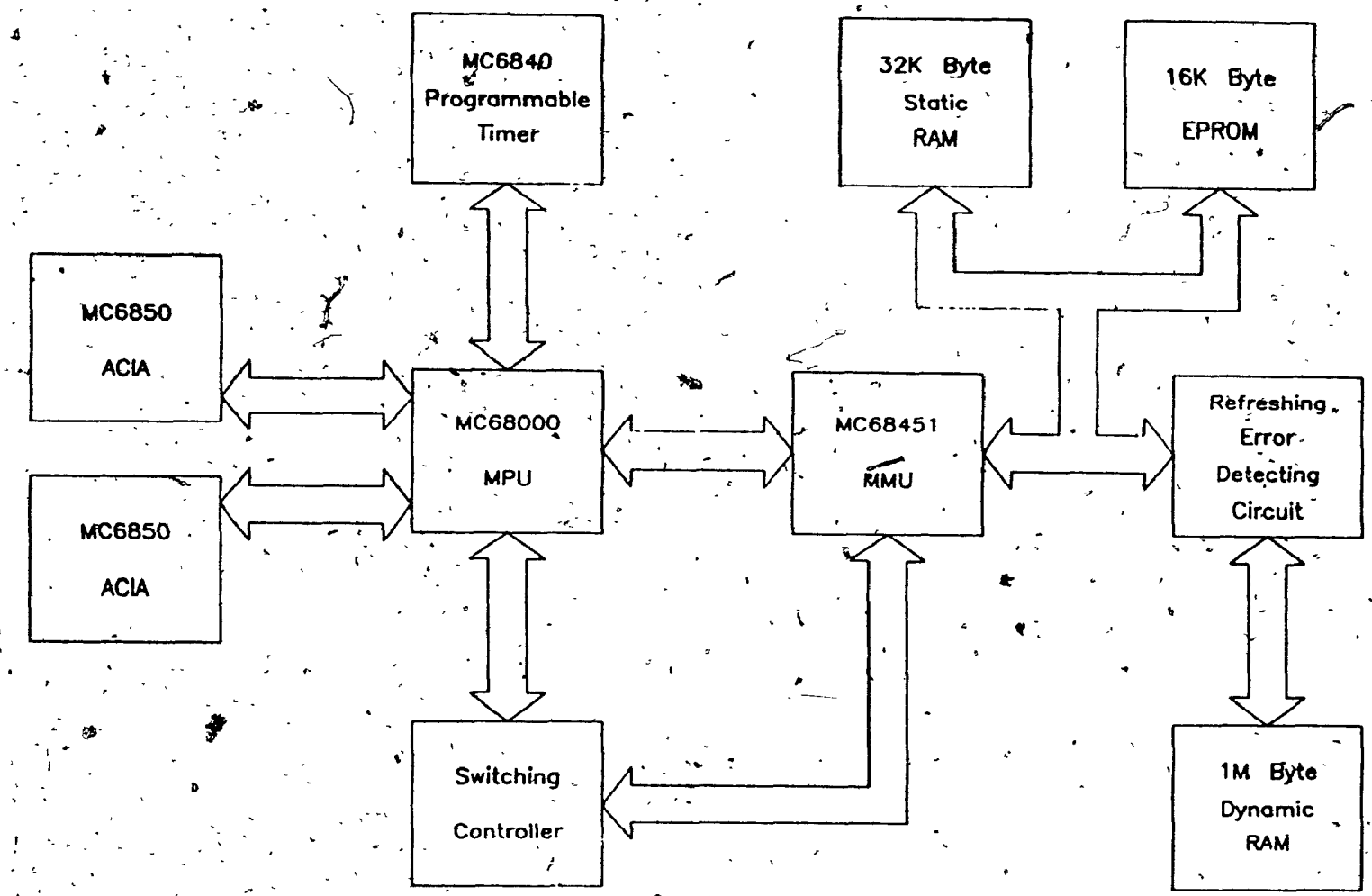


Figure 2.1 Existing Hardware of a Homogeneous Multiprocessor Node.

immediate-neighbouring processors.

The access to non-local memory is handled by the decoding logic that generates a request to a switch to close, through the addressing mechanism of a processor. The two most significant bits of the address indicate the address space which the processor is accessing. The processor is accessing its local memory module if the two most significant address bits are 00, the memory module to its left if they are 01, and the memory module to its right if they are 10. The combination 11 is reserved for local I/O. The details of the bus-access protocol and data transfer of non-local memory access are discussed in section 2.3.3.1 and 2.3.3.2.

To facilitate interrupt processing, we implemented seven prioritised hardware interrupts. In order of priority, these are RAMERR, IRQMMU, IRQTI, IRQSYS, IRQLFT, IRQRHT, and IRQACIA. The MPU recognises these interrupts from the interrupt control lines $\overline{IPL0}$, $\overline{IPL1}$, and $\overline{IPL2}$ through the encoding logic and latches. A state of 000 in the $\overline{IPL}$s indicates that there is no interrupt. The RAMERR is an interrupt from the on-board dynamic RAM and indicates an error based on the error-detecting circuit [58]. The IRQMMU responds to both the fault and interrupt requests from the MMU. IRQTI is the programmable timer (MC6840) interrupt used for timing purposes. When an incoming message is received or an outgoing message has been transmitted, the IRQSYS line is asserted and the H-Network communication module is invoked. The IRQLFT and IRQRHT are interrupts from the local processor's neighbours to be used for message passing and synchronisation among cooperating processors. This process is discussed further in section 4.1.1 on packet delivery mechanism. The IRQACIA interrupt is generated when one of the RS-232 serial lines requests service. This interrupt is to be used in conjunction with the serial line drivers described in section 3.3.3.

The switching controller is the interface between two adjacent processors. It is responsible for all the non-local memory access signals, and for arbitration in cases of deadlocks (as described in section 2.3.3.1). The switching controller is implemented in

CMOS-VLSI technology [58]. We are also in the process of implementing the controller in a Programmable Logic Array (PLA) design.

The network interface, the H-Station, is in the designing stage. However, a submodule of the H-Station that is responsible for network acquisition has been designed in CMOS-VLSI technology and sent for fabrication.

2.2. Model of the Simulator

The simulation technique used in simulating the processor array can be classified as "discrete-event simulation" where the system changes its state only at discrete moments of time [36]. Our objective is to get a snapshot of the state of the system at regular time intervals. This time-driven simulation suits our needs because all the events occurring in the system are synchronised by the common processor clock. New events take place only when the time is incremented. Therefore, the simulation snapshots at any particular point in time match closely to what will actually happen when the system is in operation.

The time interval used is one processor clock cycle†. This is the largest time interval such that events happening within it do not affect each other, and their order of execution is noncritical. The events that take place when the clock is advanced by one unit, depend on what events had happened during the previous time interval. However, old events are not altered when a new system state occurs.

This time granularity of one processor clock period establishes modularity in event simulation. In the simulator, a procedure is the implementation of a stage which represents the events that happen in a processor within a time quantum. An activity (e.g., processor instruction execution) is a collection of sequentially executing stages. A parallel program, consisting of separate procedures, is the concurrent execution of several

† Because the time quantum was chosen as one processor clock period, it is assumed that all the processors derive their clocks from the same common source

activities on several processors. On top of these procedures which define parallel activities is a monitor that advances the clock, keeps various statistical data, and serialises these activities by invoking the corresponding procedures sequentially.

The invocation of a procedure or a stage of an activity is handled by the monitoring procedure through an indirect call. As shown in Figure 2.2, the monitor fetches the corresponding pointer pointing to the next stage of an activity, and invokes the corresponding procedure by switching execution environment. These pointers are maintained by the individual stages since the invocation sequence is predictable and predetermined. This indirect call mechanism, resembling the indirect function call found in C [56], has the advantage of relieving the monitor from keeping track of the invocation sequence. Thus, additions or modifications to the simulator involve only the stages affected, and the simulator development can be handled by different programmers concurrently.

Running on a uniprocessor environment, the simulator simulates the operation of all the processors in the array, and collects meaningful statistics, both locally of a processing element and globally of the system, that are necessary for the interpretation of the behaviour of the switching network. The simulator runs on PDP-11/RMX-11, VAX-780/4.2 BSD Unix, and VAX-8600/VMS. With the exception of two small modules that implement the above mentioned indirect call mechanism written in host assembly, the rest of the simulator is written in FORTRAN 77.

2.3. Simulator Implementation

The simulator simulates the operation of an array of processors composed of k processors along with the appropriate interbus switches and local memories. The simulator currently does not handle the H-Network for which separate simulation and analytical models have been devised [68]. The individual processors simulated are 8 MHz MC68000

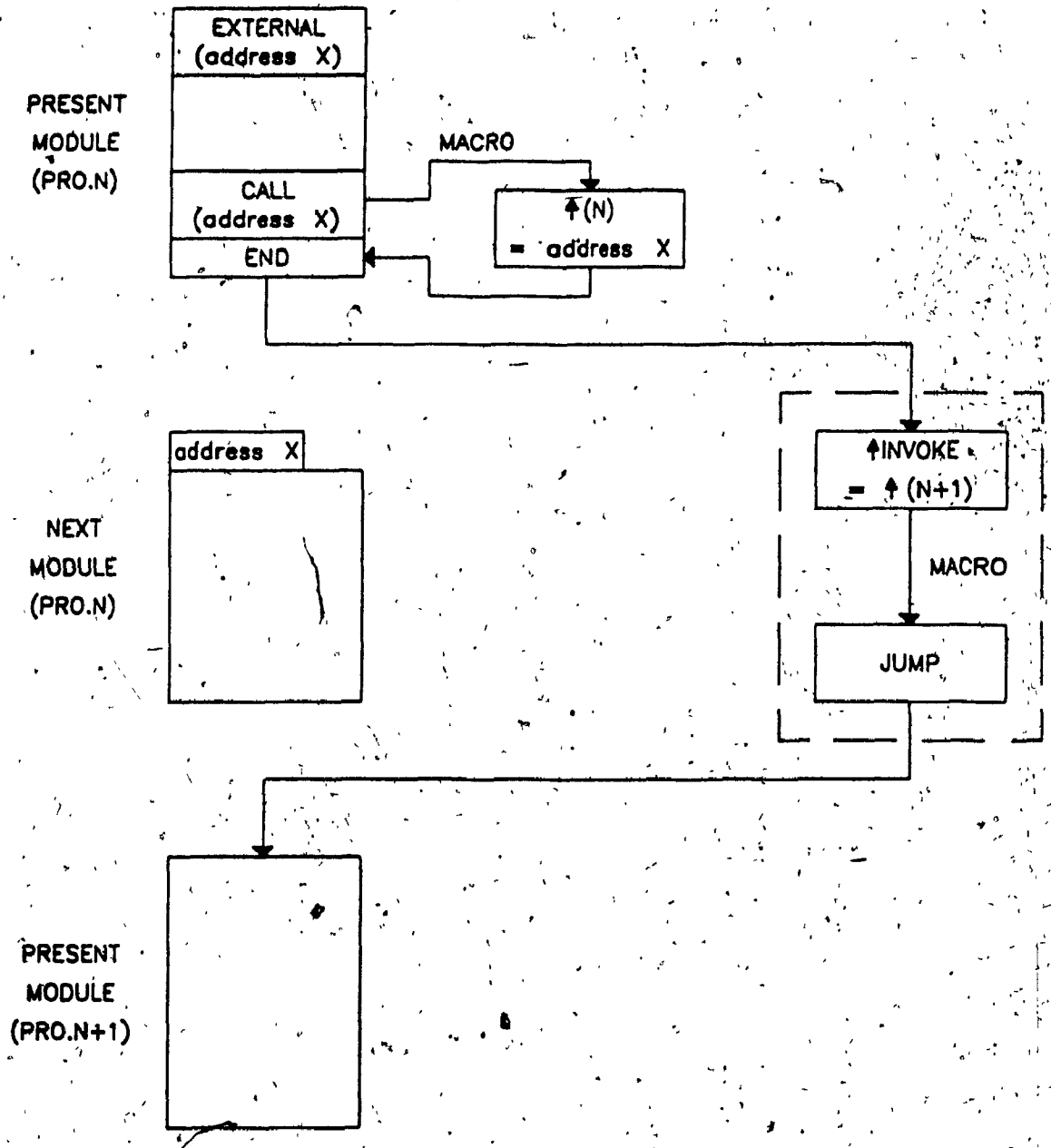


Figure 2.2 Indirect Call Mechanism.

microprocessors, which we are using as processing elements, while the simulated local memory is limited to 10 Kbytes. The maximum number of processors that can be simulated is 64. However, the amount of local memory and the number of processors can be altered for specific needs by simply recompiling the simulator.

2.3.1. Simulating the MC68000 Processor

The MC68000 microprocessor has a two-stage instruction pipeline with a two-word buffer linking the two stages [5]. The first stage prefetches instructions and immediate data, while the second one is the instruction decoding and execution unit. The two-stage instruction pipeline is depicted in Figure 2.3. The simulator simulates this pipeline by providing separate procedures for instruction prefetch and instruction execution, both of which are invoked once for each processor before the simulator clock advances. The following assumptions, however, have been made in the simulator, based on measurements performed on the MC68000 microprocessor, to account for its operational characteristics:

- The prefetch unit fetches the next instruction and immediate data continuously, and makes them available to the execution unit through the two-word buffer.
- If a memory reference (local or neighbouring) is required during the execution of an instruction, the execution unit utilises the bus directly in order to access the appropriate memory module, as long as no one else (either the local prefetch unit or a neighbour) is currently using the bus.
- When both the execution unit and the prefetch unit compete for the bus simultaneously, preference is given to the execution unit.

2.3.2. Structure of the Simulator

On the local level, for each processing element, there are two activities that proceed

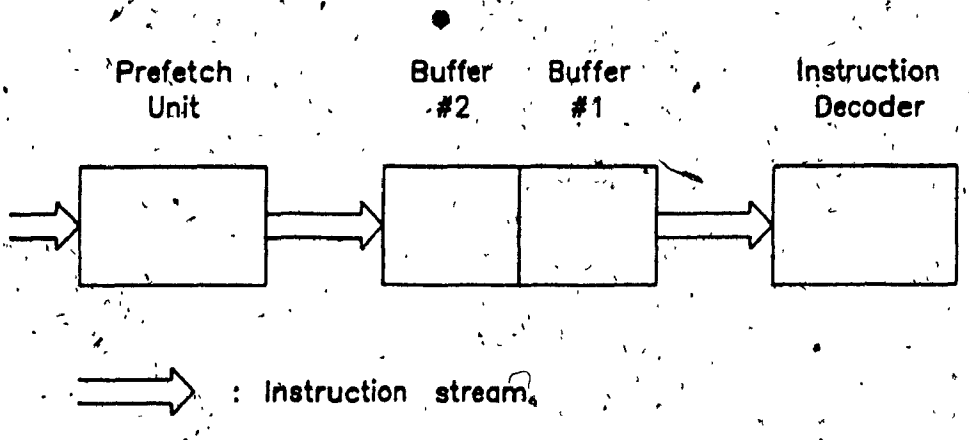


Figure 2.3, Two-stage Instruction Pipeline.

in parallel: the execution of the current instruction and the prefetch of the next instruction or immediate data. Globally, the network of interbus switches resolves any pending requests and closes the appropriate switches according to the switching algorithm (to be discussed in detail in section 2.3.3).

In the simulator, separate procedures handle the activities mentioned above, while a monitoring procedure serialises these parallel activities, maintains the clock, and collects statistics, as was discussed in section 2.2. The time granularity in the simulator is 125 ηs, which corresponds to one clock period of an 8 MHz MC68000 Each stage of activities, encapsulated in a procedure, thus represents events happened in a time interval of 125 ηs. For example, a memory reference is composed of four stages, as shown in Figure 2.4. The procedure simulating the first stage, which is the first 125 ηs of an MC68000 memory access, places the address on the address bus, and asserts the address and data strobes, according to the specification of the MC68000 operation [5].

These parallel activities, represented as stages, are invoked once within one clock period. Specifically, during one clock period, a prefetch stage is invoked for each simulated processor starting with the leftmost one, then an instruction execution stage, and finally an interbus switching stage—again for each simulated processor starting with the leftmost one—are invoked. Figure 2.5 shows the order of these activities for a system comprising of three processors. There are three pointers, used for the indirect call mechanism, for each processing element simulated. Each pointer is associated with one of the three activities mentioned above, i.e., prefetch, instruction execution, and interbus switching.

2.3.3. Switching Algorithm

The simulator models the actual operation of the switching system, where the bus-switching process involves two phases, and is simulated as such. The switching algo-

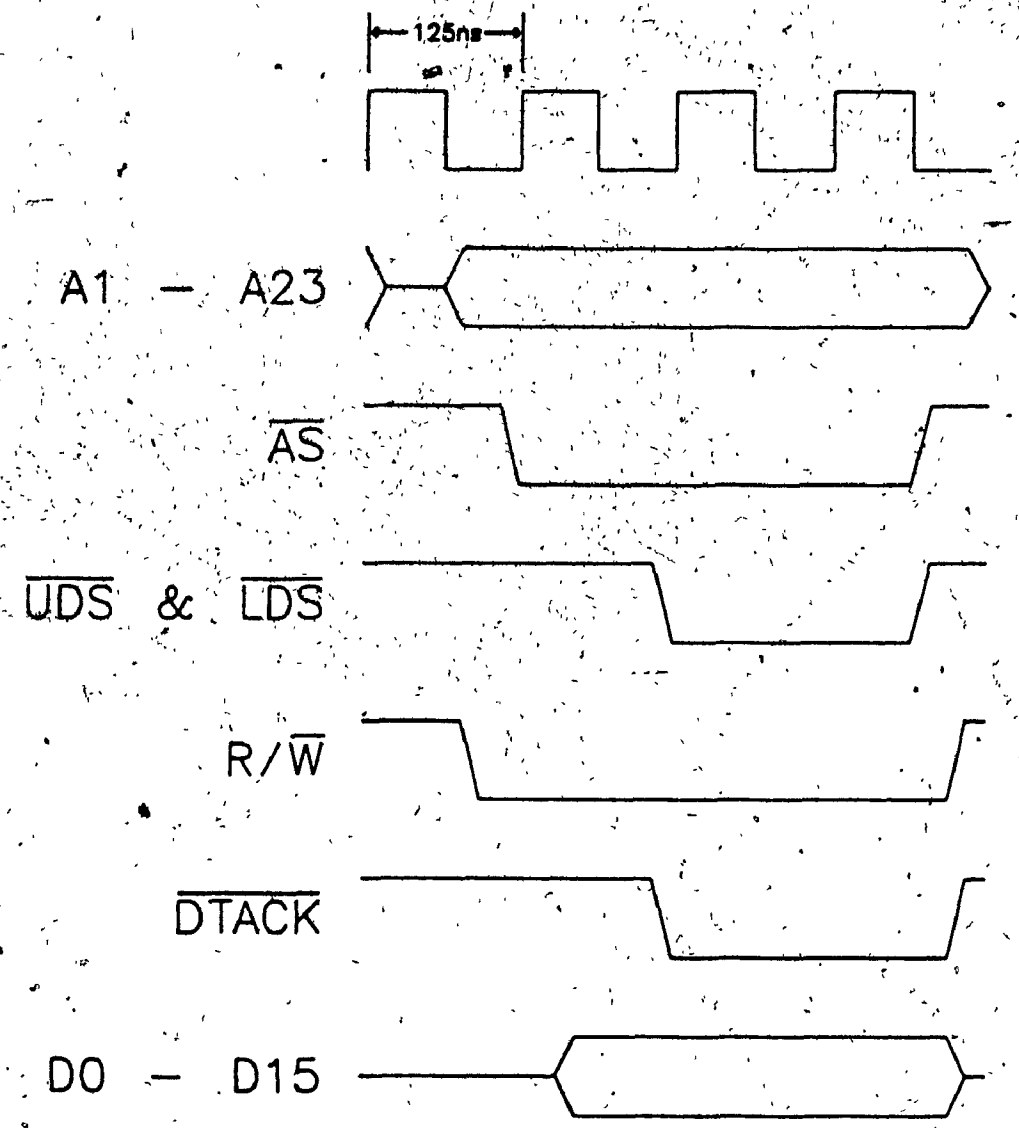


Figure 2.4 Write Cycle Timing.

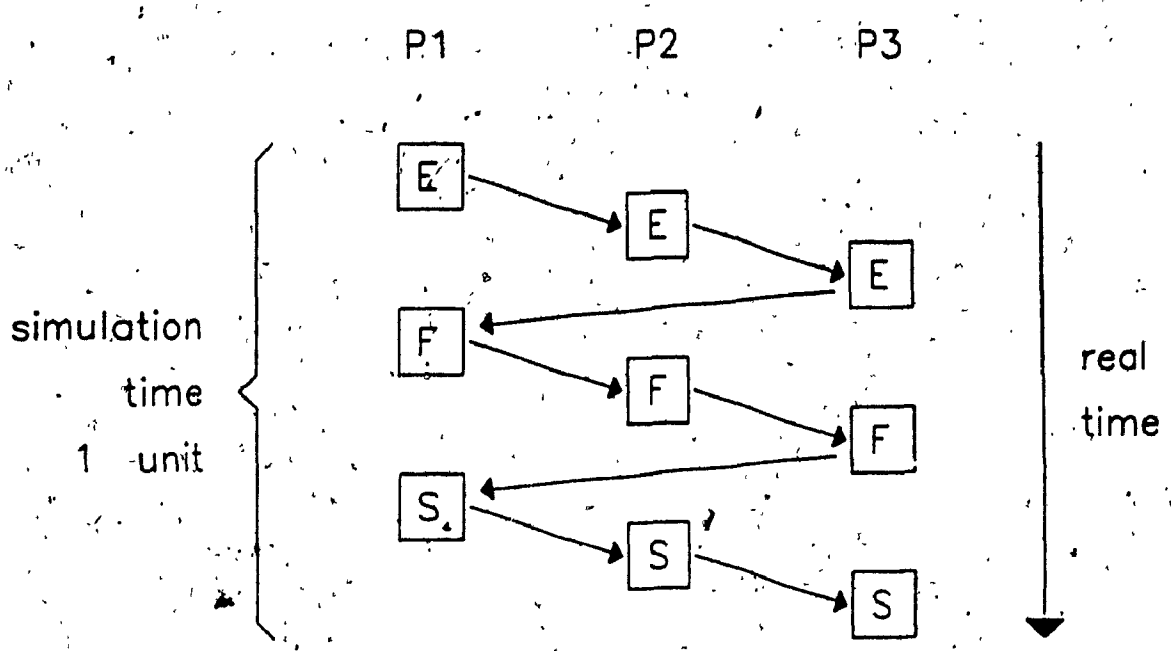


Figure 2.5 Parallel Activities within One Simulated Clock Period Comprising 3 Processors.

→: simulation path
E: instruction execution
F: instruction prefetch
S: switching algorithm

rithm shown in Figure 1.2 is used for the processor-independent first phase of the process. A switch is said to be logically closed as a result of the execution of the switching algorithm. The simulator uses the fact that the implementation processor is an 8 MHz MC68000 in order to implement the processor-dependent second phase of the bus-switching process, where a switch is physically closed only when the connection of the two neighbouring buses is established. Thus, a distinction is made between the events of logical and physical closures of a switch in discussing the design of the simulator.

2.3.3.1. Phase 1: The Logical Closure of the Switches

The switching algorithm handles the first phase of a switch closure, which is implemented in hardware with a Very Large Scale Integrated (VLSI) circuit. The switching network, composed of individual switching hardware modules placed between processing elements, is realised as a synchronous sequential circuit with a common clock.

It is anticipated that this global clock, driving the switching network, is much faster than the clocks of the individual processors (~25 MHz as compared with 8 MHz for the MC68000 clocks). A stable state of the switching network is reached after a maximum of three iterations of the switching algorithm, reflecting the maximum number of transitions (Open—Gray—Closed) a switch may undergo. The faster clock rate thus stabilises the state of the switching network within one processor clock period from any change in the switching module's inputs. In the simulator, the switching algorithm is therefore executed three times so as to ensure a stable state of the switching network at the end of each clock cycle.

Once a switch is in a Closed state, as determined by the switching algorithm, it only indicates that the switch is logically closed. The actual closing of the switch, and the arbitration in cases where both processors adjacent to the switch have pending requests, are left to the second phase of the switch closing operation.

2.3.3.2. Phase 2: The Physical Closure of the Switches

The switching algorithm ensures only the safe and live operation of the switches. At the end of Phase 1, the switching algorithm has decided whether a switch can be closed, the actual closure has yet to be accomplished. A physically closed switch connects two adjacent buses, thus permitting a processor to access the memory residing on the bus of one of its neighbours. Therefore, during the time that a switch is closed, the neighbouring processor must be isolated from its own bus. This is a necessary precondition for a switch to actually close, which can happen either immediately by floating the bus drivers of the requested processor or after a normal bus request cycle. The immediate floating of the bus drivers is enforced only after arbitration and is used in order to break deadlocks that may occur in the switch closing process.

The MC68000 uses a three-signal bus-arbitration protocol [5]. A requesting device issues a bus request signal (BR) to the processor. The processor asserts a bus grant (BG) immediately after internal synchronisation. The BG indicates that the processor will yield the bus after the completion of the current bus cycle. The requesting device waits until the end of the current bus cycle, then asserts the bus grant acknowledge (BGACK), negates bus request, and becomes master of the bus The processor, upon detection of the negated bus request, negates bus grant and waits for the negation of bus grant acknowledge, at which time the requesting device has completed its data transfer operation.

Our bus request protocol is a two-signal arbitration protocol consisting of a bus request signal (BR) originating from a switch to a processor, and a bus grant signal (BG) generated by a processor at the end of its current bus-access cycle. The asynchronous timing of these signals follows closely the timing of the more general three-signal bus-arbitration protocol found in the MC68000. The generation of a bus grant by a processor indicates that it has taken itself off its local bus and, therefore, the bus can now be utilised safely by a neighbour.

In the Motorola asynchronous data transfer protocol, a processor accesses resources (e.g., memory) on a local or extended bus by asserting the address strobe (AS) signal. At the end of the data transfer cycle, a data transfer acknowledge (DTACK) signal is issued by the resource.

In our asynchronous bus-access protocol, the request to a switch to close is generated through the addressing mechanism of a processor, with the assertion of address strobe. If a processor needs to access an item located in one of its two neighbouring memory modules, it addresses it with the two most significant bits of the address being set according to the convention discussed in section 2.1. The requesting processor then waits for a data acknowledgement (DTACK, generated by the addressed device, e.g., memory module or peripheral) in order to complete the access sequence. The corresponding switch interprets the request and at the end of Phase 1, generates a bus request to the appropriate neighbouring processor. Once the bus is granted, the switch closes and an extended bus is created through the completion of the memory access cycle, until the detection of the DTACK signal.

The switching algorithm works well in determining the closure of a switch in a logical point of view. However, a deadlock may occur in reality, when the actual hardware signals are taken into consideration. For instance, if two adjacent processors request the service of their in-between switch, and at the end of Phase 1 the switch is logically closed, both will wait for an acknowledgement, normally granted by a neighbouring processor at the end of its current memory access cycle. But if the neighbouring processor is waiting for an acknowledgement to complete its current memory access sequence, there is a deadlock as both processors are involved in a deadly embrace situation.

There are three possible cases of deadlock, as illustrated in Figure 2.6. The switching hardware implemented, as well as the simulator, can detect these deadlocks and break the deadlock by giving the priority to use the switch to one of the processors. The three deadlock cases are:

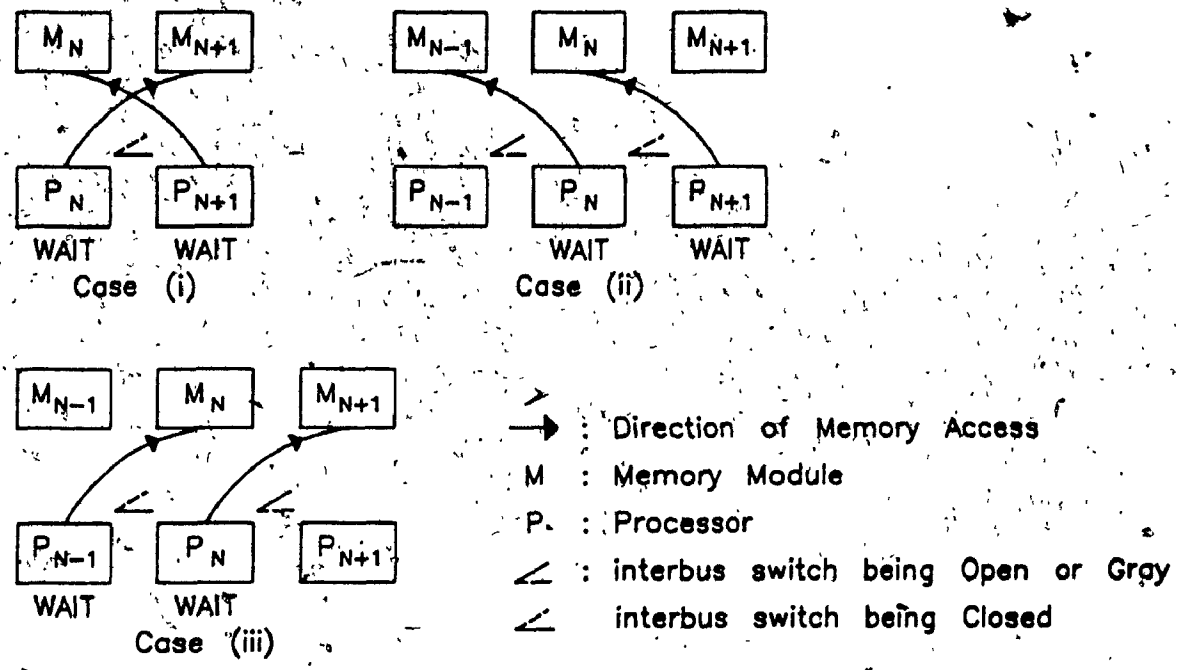


Figure 2.6 : The Three Possible Deadlock Cases.

Case 1: Two neighbouring processors request simultaneously (within the same processor clock cycle) their in-between switch to close. As stated in the asynchronous bus-access protocol discussed earlier, both processors would wait for a data transfer acknowledgement before they terminate their current cycle. The requests for the switch to close are received by Phase 1, which eventually will put the switch in a logically closed state. The second phase, in order to complete the switch closure, will request the bus from the two processors and wait for a bus grant signal, issued by one of the requested processors at the end of its current bus-access cycle. The bus grant signal, however, will never materialise since the requested processors are themselves in the middle of a bus-access cycle and are waiting for the switch to close physically in order to complete the access sequence. In this case, the hardware breaks the deadlock by granting the bus to the processor located to the right of the switch. This is achieved by forcing the bus drivers of the processor to the left of the switch to float, and generating a bus grant signal to the favoured processor.

The other two cases involve two adjacent processors which request neighbouring memory modules through two adjacent switches.

Case 2: Two adjacent processors (e.g., processors P_i and P_{i+1}) attempt to access the memory module of their left neighbours (M_{i-1} and M_i respectively). Both processors initiate a bus-access cycle and issue requests to the appropriate switches (S_{i-1} and S_i) to close. If the request from processor P_{i+1} arrived at the switch S_i at the same time as or earlier than the request from processor P_i to the switch S_{i-1}, then, according to the switching algorithm, switch S_i will eventually Close, while switch S_{i-1} will remain Open or Gray. The second phase of the Closed switch S_i will now attempt to gain mastership of processor P_i's bus by issuing a bus request and waiting for a bus grant from P_i. The bus grant signal, however, will never be issued by processor P_i since itself is in the midst of a bus-access cycle, which is atomic and which can only be completed when switch S_{i-1}

closes. But switch S_{i-1} cannot close, even logically, as long as switch S_i is Closed, hence, a deadlock. Once this deadlock situation is detected, both the hardware and the simulator break the deadlock by granting the bus to processor P_{i+1}, and forcing the bus drivers of processor P_i to float.

Case 3: Two adjacent processors (e.g., processors P_{i-1} and P_i) attempt to access the memory modules of their right neighbours (M_i and M_{i+1} respectively). Both of them initiate a bus-access cycle and issue requests to the appropriate switches (S_{i-1} and S_i) to close. If the request from processor P_{i-1} arrived at the switch S_{i-1} earlier than the request from processor P_i to the switch S_i, then, according to the switching algorithm, switch S_{i-1} will eventually Close, while switch S_i will remain Open. Similar to Case 2, a deadlock occurs. In this deadlock situation, the bus is granted to processor P_{i-1}.

These deadlock cases will eventually complete the handshaking process through arbitration. All other non-deadlock cases are handled through a normal bus request cycle.

In the simulator, the bus request and data transfer protocols were implemented as described, however, some timing assumptions have been made concerning propagation and processing delays, based on the projected hardware implementation. If the resource is located on the processor's local bus, then the access is to be completed in exactly four clock periods (500 ηs). If the resource is on a neighbour's bus, then an extended bus must be created. A switch will logically close, at the earliest, one clock period after it has received the request to close, following the execution of the switching algorithm. Immediately after the logical closure, the switch will generate a bus request to the appropriate neighbour and will wait until a bus grant is received before it physically closes. If arbitration is necessary to break any deadlock, the switch will close immediately according to the discussion above; otherwise the requested processor will latch the bus request two clock periods after this signal has been received and will generate a bus

grant (and also take itself off its local bus) at the end of its current bus-access cycle. The switch, once it detects the bus grant signal, will physically close in one clock period. This delay in closing the switch physically is introduced in order to account for the various signal propagation delays. Figure 2.7 gives all the possible cases for handshaking timing of a bus request signal relative to an ongoing bus-access cycle of a neighbouring processor. From this figure, the minimum and maximum delays in forming an extended bus (after a switch has logically closed and a bus request has been generated) can be calculated. The projected minimum access time, if the switch is logically closed immediately, is 8 clock cycles, while the maximum access time is 11 cycles.

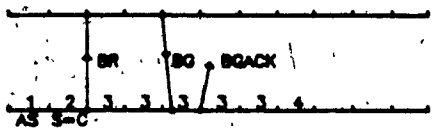
The simulator has been designed and implemented with key features as were described in this section. Appendix B displays the user interface and monitoring program while a sample module of the simulator showing the stages of an instruction is included in Appendix C. Appendix D shows the list of instructions simulated. The input and output specification for the simulator can be found in Appendix E.

2.4. Simulation Experiments

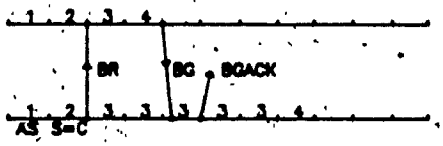
We have performed several simulation experiments through which we were able to establish a measure of the performance of the Homogeneous Multiprocessor [19,20,42]. But before any performance results were analysed, we made a check on the accuracy of the simulator.

A subset of the MC68000 instructions is implemented in the simulator. The simulated execution time of each of these instructions was compared with the actual execution time and they matched exactly. We expected, therefore, the simulated execution time of an instruction stream, consisting of different instructions, must also match with the actual one.

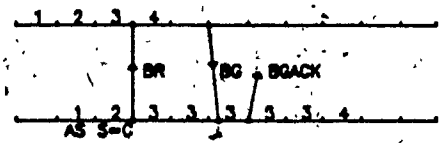
Operation timing results from the simulation experiments, pertaining to a single



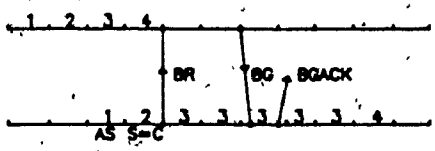
Case (i) access time – 8 periods



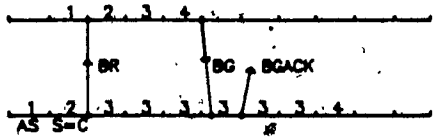
Case (ii) access time – 8 periods



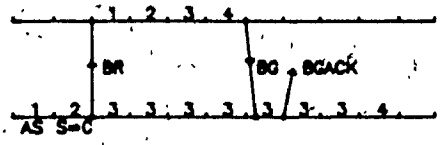
Case (iii) access time – 8 periods



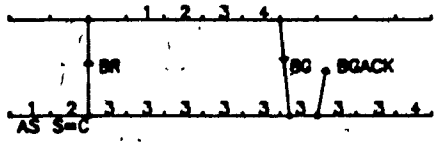
Case (iv) access time – 8 periods



Case (v) access time – 9 periods



Case (vi) access time – 10 periods



Case (vii) access time – 11 periods

Figure 2.7 Bus request cycle timing. The upper trace represents the requested processor. The lower trace represents the requesting processor. Each slot represents one clock period. Slot Number 3 is essentially a wait.
AS: address strobe.
BG: bus grant.
BGACK: bus grant acknowledge.
S=C: the intervening switch is logically closed

processing element in the calculation of autocorrelation functions, were compared with actual performance data as obtained from the operation of a prototype node and is shown in Table 2.1. The simulator behaves accurately as compared to the actual hardware. The deviation in time execution from the actual operation stays within bound of 0.014 percent error. Within the limit of the projection on switching delays, we felt that the simulator is a viable tool to study the Homogeneous Multiprocessor.

The simulation experiments can be divided into two classes. The first class establishes the performance of the Homogeneous Multiprocessor as it is affected by the demand of the network of switches, while the second class consists of simulation experiments which simulated a distributed algorithm for finding the autocorrelation functions of a given signal.

2.4.1. Performance Study under a Varying Frequency of Requests to the Network of Switches

The aim of these experiments was to study the delays incurred in accessing resources of a neighbour and thus to evaluate the operation of the switching network. This was accomplished through the execution, on each processor, of a simple program consisting of a DO loop, during which one reference to a neighbouring memory module was made. By varying the length of the DO loop, we were able to obtain varied frequencies of requests to the switching network. The range of requests varied from the most intensive case, where all processors are fetching instructions and data from their neighbours, to the least intensive case, where only one processor is allowed to access a neighbouring memory module. Also, by varying the relative location of the requests to the neighbours within each loop, we were able to control the interarrival intervals of the requests as they were presented to the switching network.

Two scenarios were chosen in the pattern of requests. For the first, all the refer-

Table 2.1 Simulation and Actual Timing Comparison

n : number of autocorrelation functions

n	simulation time	actual time	percentage error
2	1.39749e-02	1.39750e-02	8.93003e-03
4	1.84611e-02	1.84620e-02	4.74187e-03
6	3.19199e-02	3.19230e-02	9.79081e-03
8	4.08924e-02	4.08970e-02	1.13133e-02
10	4.98649e-02	4.98710e-02	1.22879e-02
12	5.88374e-02	5.88450e-02	1.29526e-02
14	6.78099e-02	6.78190e-02	1.34578e-02
16	7.67824e-02	7.67870e-02	6.02551e-03
18	8.57549e-02	8.57670e-02	1.41424e-02

ences were made to the memory modules of the left neighbours, while, for the second, all the references were made to the modules of the right neighbours. These scenarios were chosen in order to study the effect of the preferences introduced by (i) the arbiter during the resolution of the deadlock cases, and (ii) the switching algorithm. All the memory accesses for these experiments took place to either the neighbouring memory modules, or the local memory modules, except of the accesses requested by the rightmost processor in the first scenario, and the leftmost processor in the second scenario, which always accessed their own memory modules.

For these experiments, we simulated a system consisting of ten processors, and we measured the following parameters: the average memory access time in order to make a neighbouring memory reference; the average idle time for a processor, defined as the time the processor is forced to idle because its local bus is being used by a neighbour; and the average wait time for a processor, defined as the time a processor must wait because the bus of its neighbour is not available. The experiments were performed several times and measurements for each frequency of requests were obtained based on the average of 200 experiments. Typical programs simulated in individual processors can be found in Appendix F.

The variation of these parameters as a function of the probability of a neighbouring memory access are given in Figures 2.8, 2.9 and 2.10. Two observations can be made from these plots. First, the interarrival time of the requests affects the performance of the network of the switches. This was expected, since, depending on the interarrival time of the requests, a varying number of consecutive switches may become Gray at the same time. The larger the number of consecutive Gray switches, the longer the average memory access time, since in such a case the leftmost Gray switch must wait for all the other switches in the sequence to close before it can close itself. Nevertheless, the length of these Gray sequences remains small, as can be seen from the magnitude of the standard deviation as included in Figure 2.8. Secondly, there are small differences between

the performances corresponding to the two scenarios. These differences can be attributed mainly to the arbiter and the switching algorithm. For the most intensive cases as shown in Figure 2.11, these differences are more evident as the average memory access time to a neighbouring module is plotted as a function of the number of processors, varying between two and twenty. We felt that the differences in these experiments can be attributed to synchronisation, referring to the situation where a great number of switches become Gray at the same time, and the pattern repeats itself throughout the experiment. However, these are limited cases, where all processors execute code located in their neighbours' memory modules. In actual programming practice, such a situation is not likely to arise as an experienced programmer will elect to write code that will execute within the local memory. Nevertheless, even under such an extreme loading, the average memory access time remains bounded and less than 2.2 μs. The idle time also remains within acceptable limits and never exceeds 30 percent of the total processing time. The idle time is a good indicator of the interference by a processor requesting resources from a neighbour. The fact that the idle time, even in the most demanding cases, remains within bounds means that the interference of a processor trying to use the resources of a neighbour is minimum. Also notable is the absence of any congestion in the results, as shown in the figures.

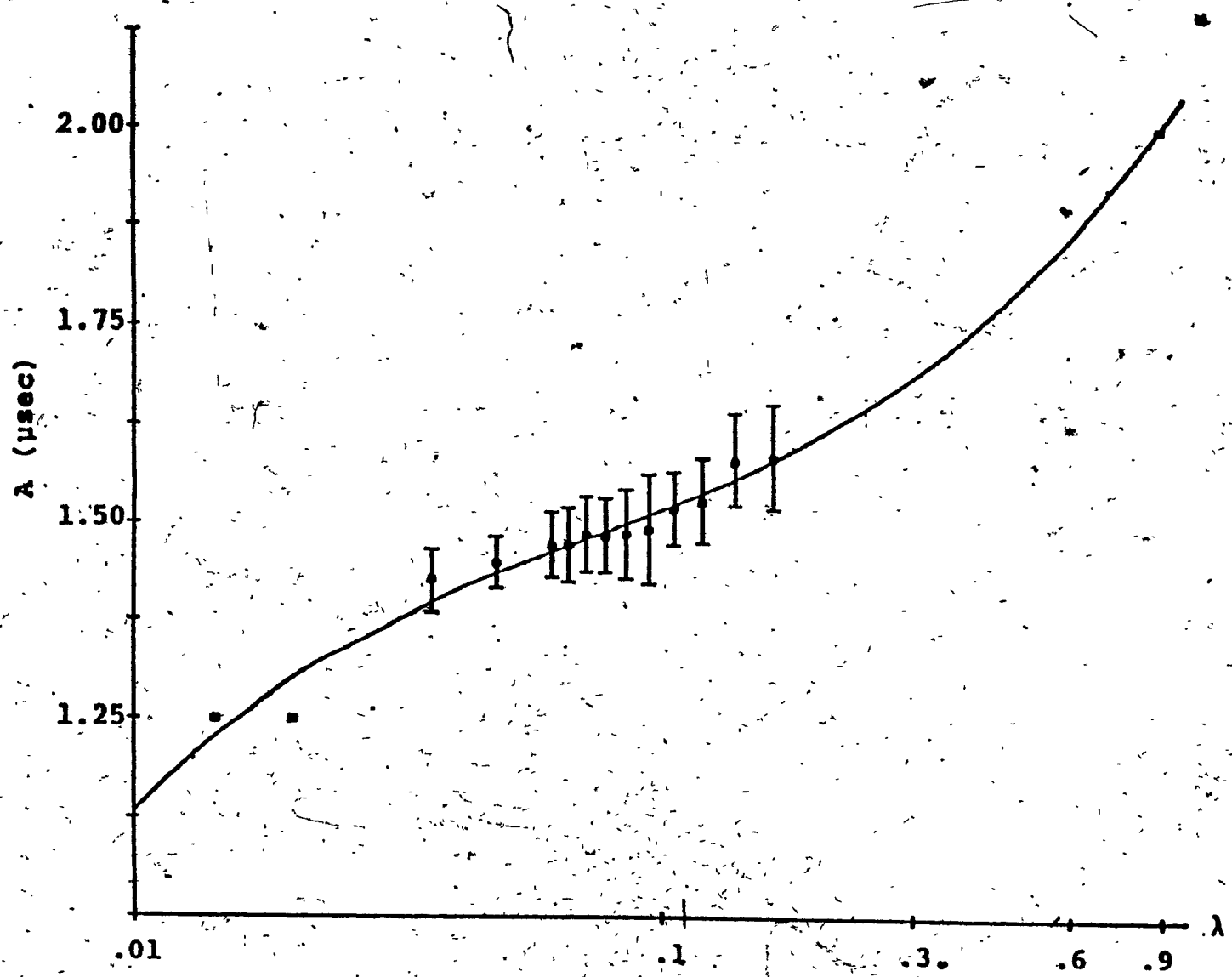


Figure 2.8a. Average Memory Access Time A for neighboring memory modules (requests to right), vs. the ratio λ, of neighboring memory accesses per memory references.

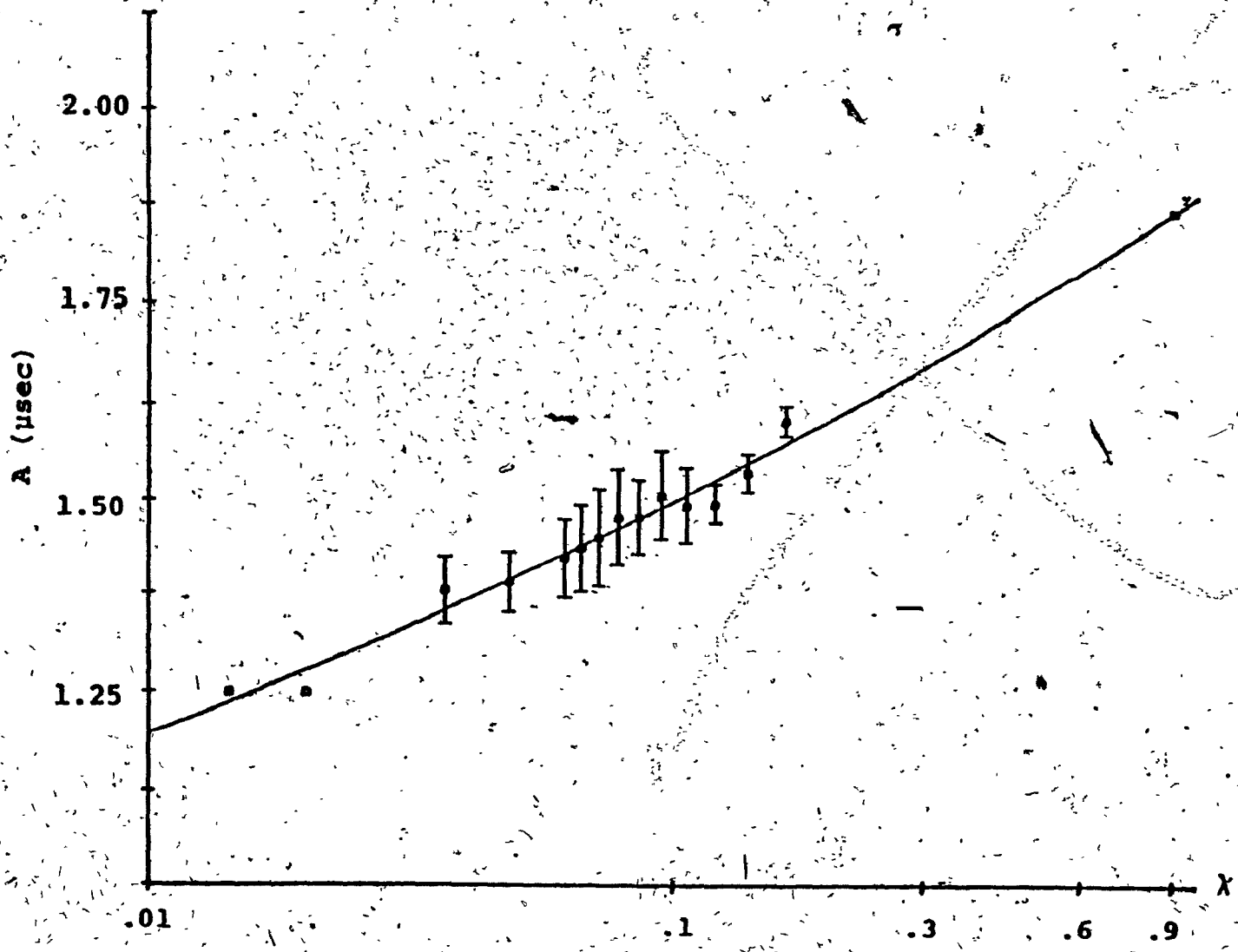


Figure 2.8b. Average Memory Access Time A for neighboring memory modules (requests to left), vs. the ratio λ, of neighboring memory accesses per memory references.

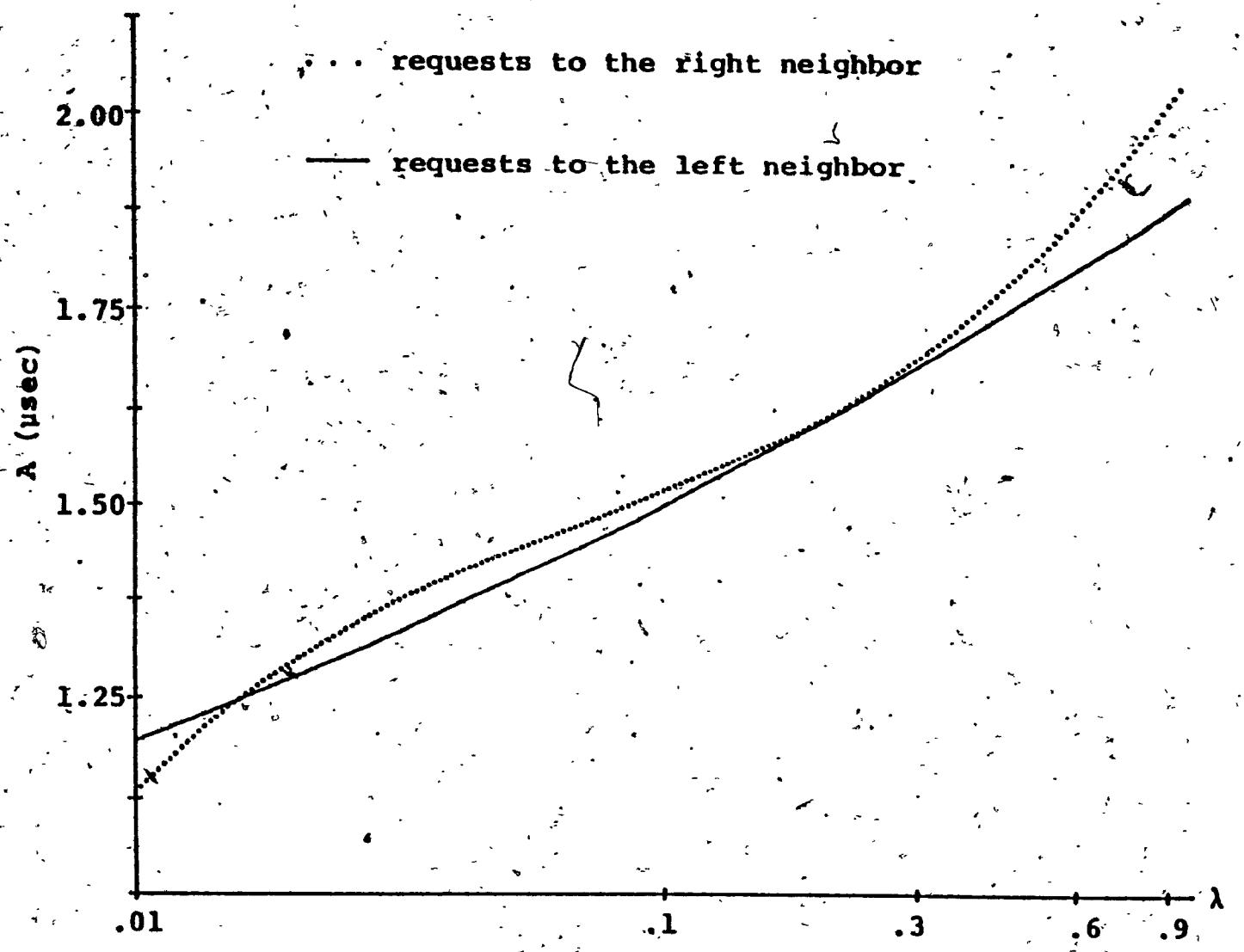


Figure 2.8c. Average Memory Access Time A for neighboring memory modules, vs. the ratio λ, of neighboring memory accesses per memory references.

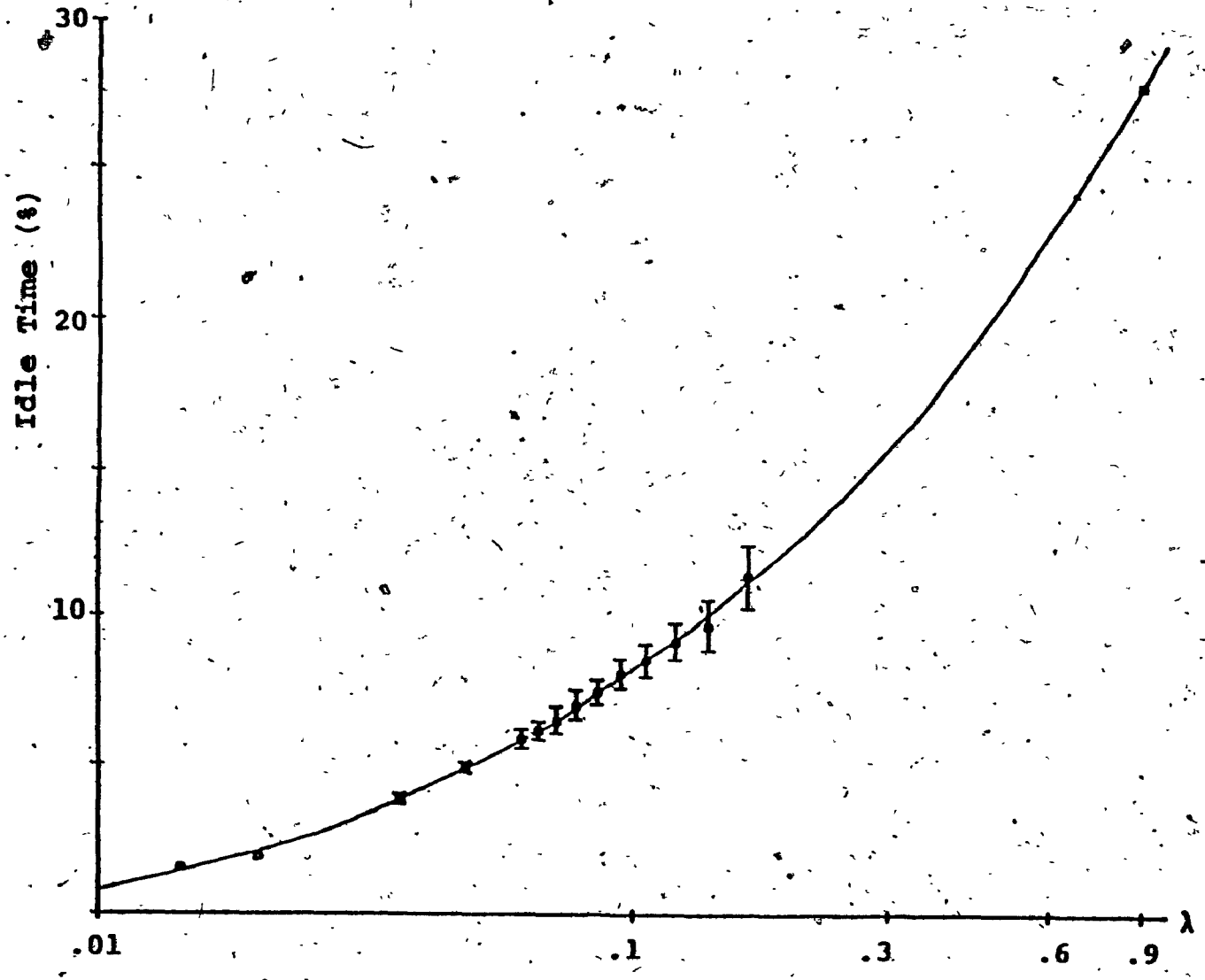


Figure 2.0a. The Idle Time (expressed as a percentage of the total simulation time) vs, the ratio λ, of neighboring memory accesses (requests to right) per memory references.

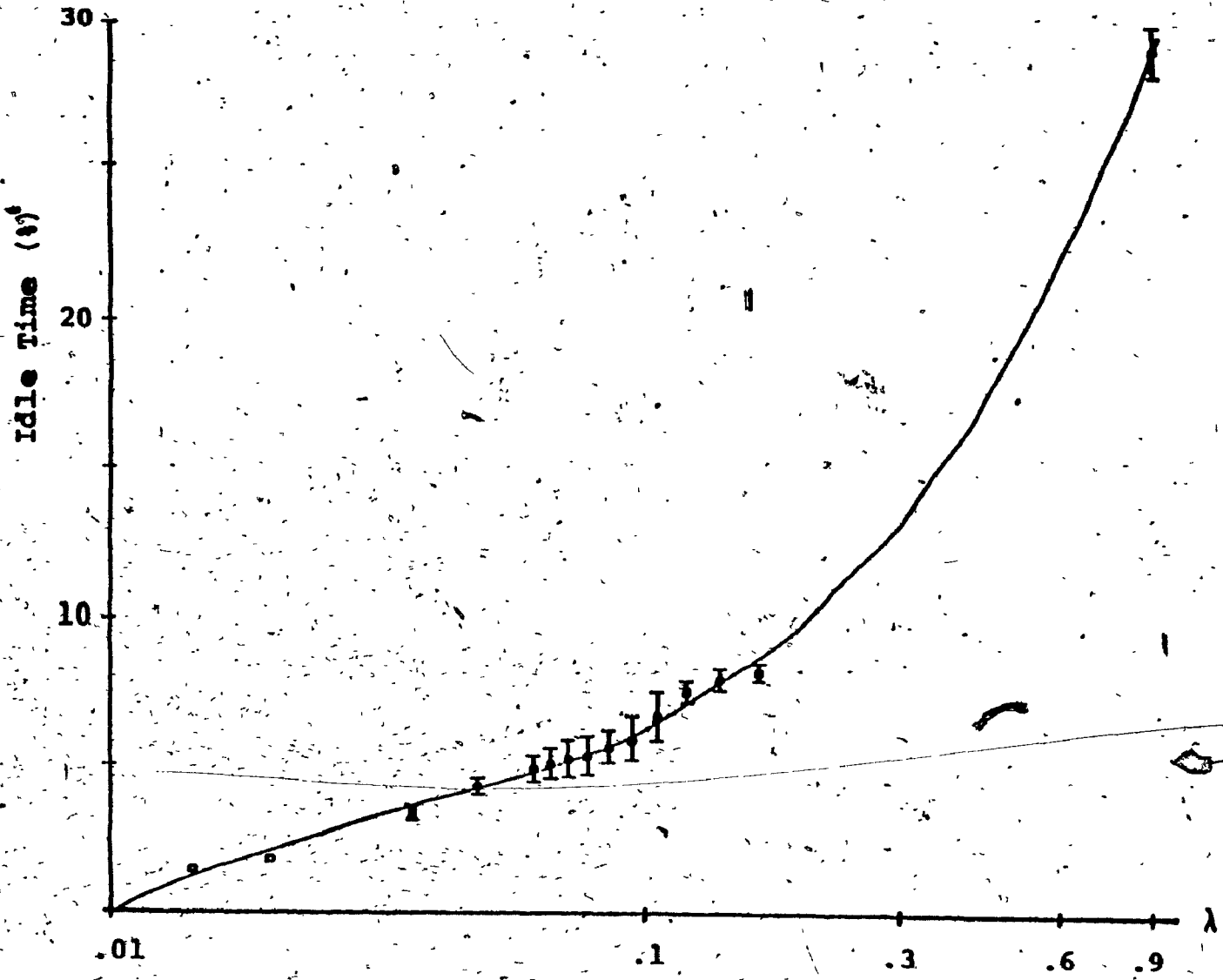


Figure 2.9b. The Idle Time (expressed as a percentage of the total simulation time) vs. the ratio λ, of neighboring memory accesses (requests to left) per memory references.

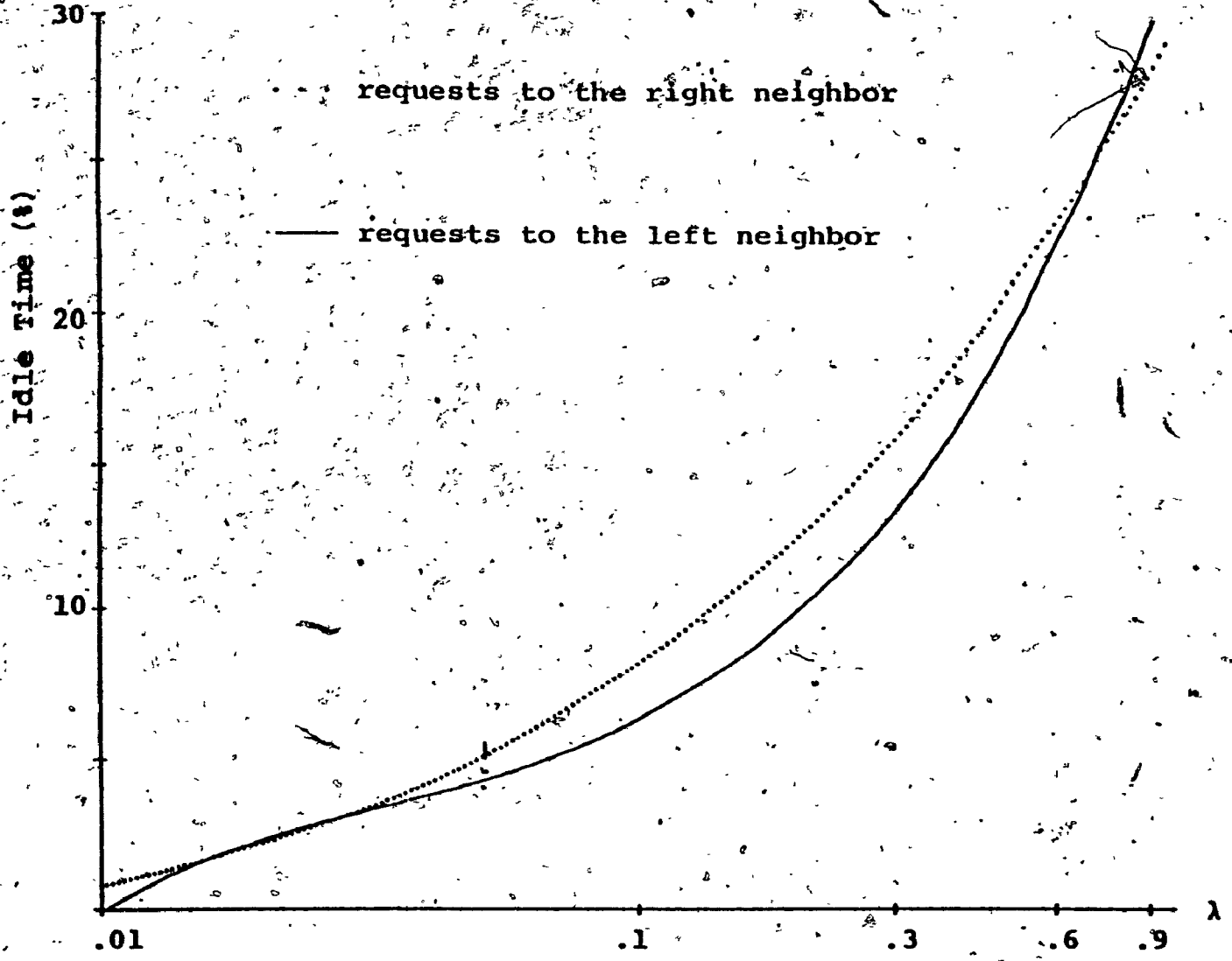


Figure 2.9c. The Idle Time (expressed as a percentage of the total simulation time) vs. the ratio λ, of neighboring memory accesses per memory references.

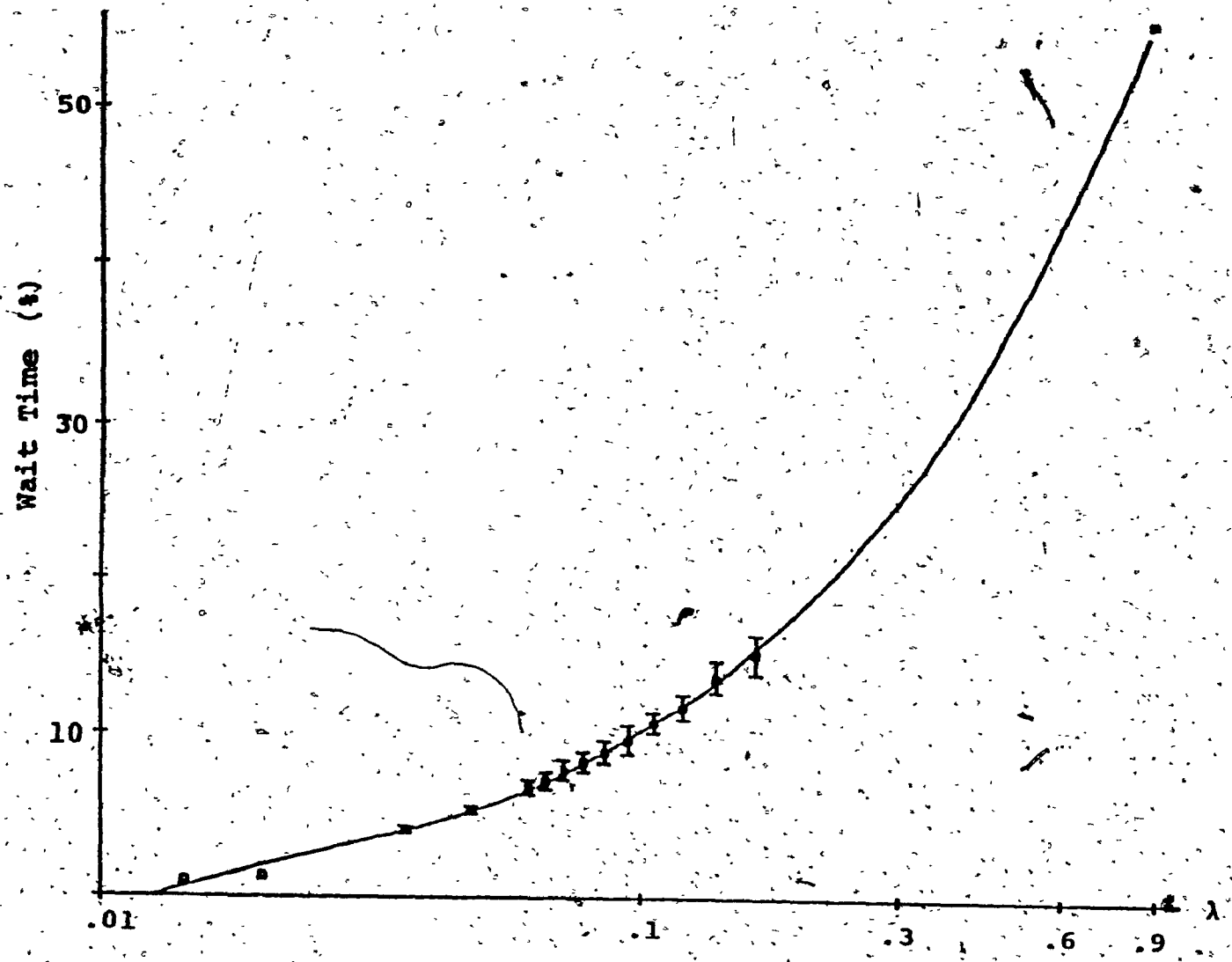


Figure 2.10a. The Wait Time (expressed as a percentage of the total simulation time) vs. the ratio λ, of neighboring memory accesses (requests to right) per memory references.

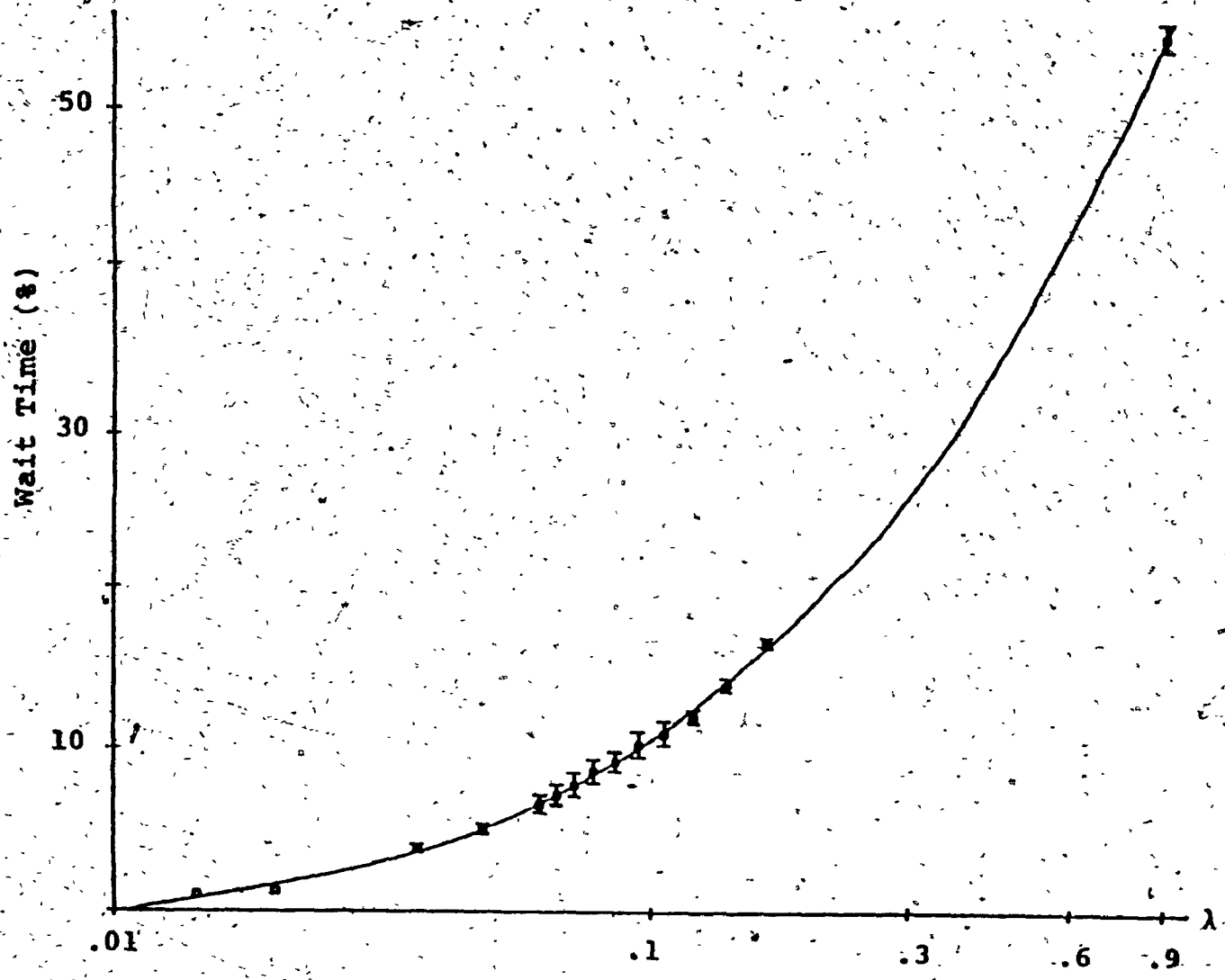


Figure 2.10b. The Wait Time (expressed as a percentage of the total simulation time) vs. the ratio λ, of neighboring memory accesses (requests to left) per memory references.

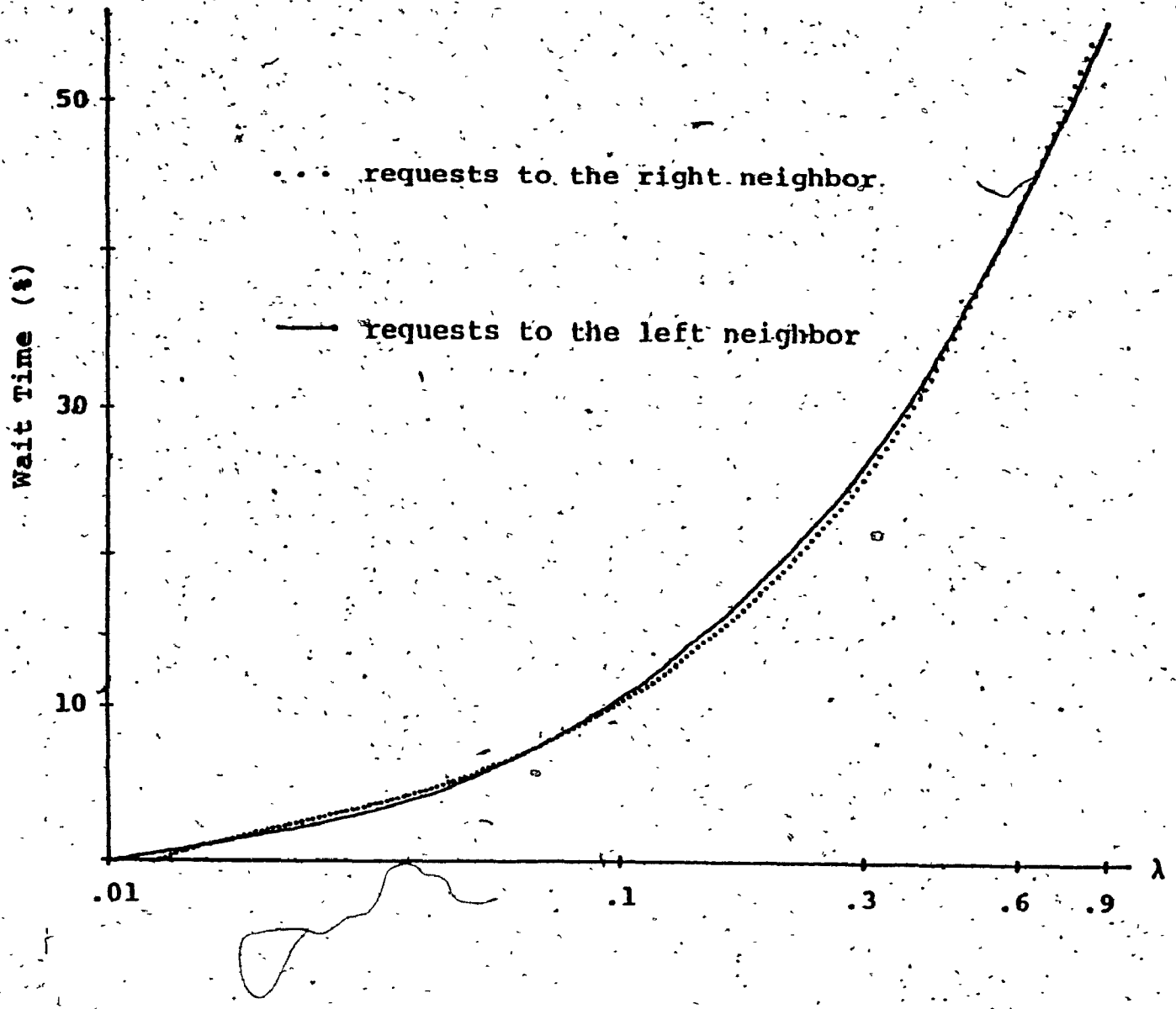


Figure 2.10c. The Wait Time (expressed as a percentage of the total simulation time) vs. the ratio λ, of neighboring memory accesses per memory references.

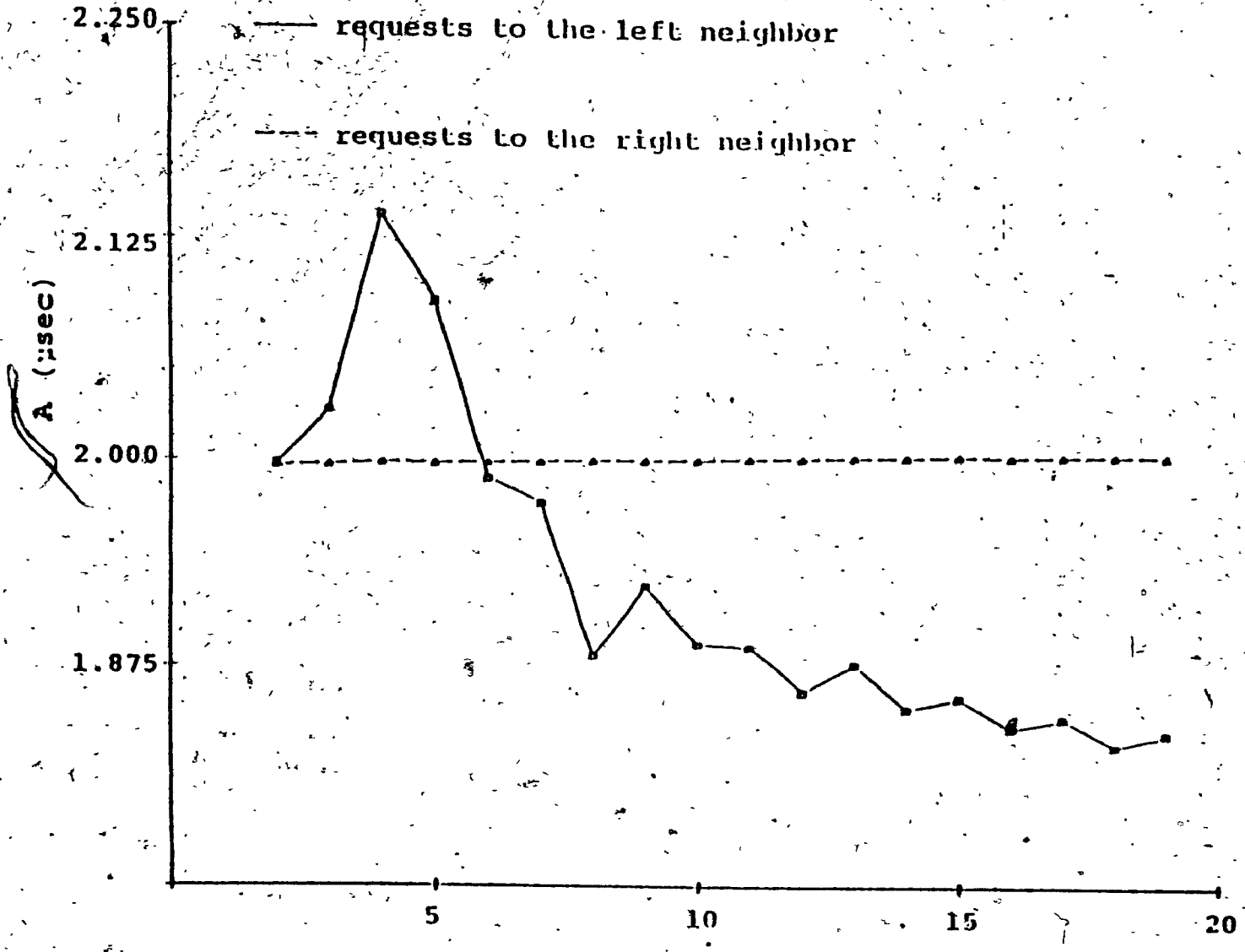


Figure 2.11. Average Memory Access Time vs. the number of processors (All the processors access the memory modules of their neighbors all the time).

2.4.2. Performance Study for the Execution of a Distributed Autocorrelation Algorithm

As a general study of how efficient the Homogeneous Multiprocessor would be in executing parallel programs, we have simulated a distributed implementation of an autocorrelation functions calculation. The autocorrelation functions R_k of a signal s_i, $i=0,1,\ldots,n$ are defined as follows [66]:

$$R_k = \sum_{i=0}^{n-k} S_i S_{i+k} \qquad \ldots(1)$$

In a uniprocessor environment, the algorithm calculating equation (1) can be described as:

```
program autocorrelation (input, output);
  const n=25; d=18;
  var R: array [0..d] of real;
    s: array [0..n] of real;
    i,k: integer;
  begin
   for k:= 0 to d do
     begin
       i:=0;
       while (i+k <= n) do
         begin
           R[k]:=R[k]+s[i]*s[i+k];
           i:=i+1;
         end;
     end;
  end.
```

In a distributed environment, the calculation of R_k may be decoupled, and single processors can be assigned to the calculation of each of the autocorrelation functions R_k. The data can be fed synchronously to all the processors through the H-Network, or be passed sequentially from a neighbour to a neighbour. In this simulation we have adopted the second alternative (passing the data from a neighbour to a neighbour), since such an approach can be easily adapted to a systolic array implementation. The overall structure of the computation is given in Figure 2.12.

Each processor calculates one of the autocorrelation functions, the leftmost one calculating R_d and the rightmost one calculating R_0. Data is passed from left to right. The leftmost processor (which takes the incoming data s_i, $i=0,1,...,n$) maintains a queue of length d and calculates the d^{th} autocorrelation function R_d by forming the sum of the products of the incoming s_i and the stored s_{i-d}, then it passes s_i to the next processor, which maintains a queue of length $d-1$ and calculates the function R_{d-1}. The process is repeated by all the processors in the array. The rightmost processor thus has a FIFO of length 0 and calculates the function R_0. We have used data sample sizes of 256 elements and 1024 elements to simulate the calculation of up to 19 autocorrelation functions.

In these simulation experiments, we have measured the efficiency of the multiprocessor approach through the speed-up factor S_k defined as:

$$S_k = \frac{TU_k}{TM_k}$$

where TU_k is the simulated time needed in order to complete the calculation of k autocorrelation functions on a single MC68000 processor and TM_k is the time needed in order to complete the same calculation on the Homogeneous Multiprocessor. The dependence of the speed-up factor on the number of autocorrelation functions obtained is given in Figure 2.13. In the figure, the lower two curves represent the speed-up factors obtained through simulation, while the upper one is the theoretically expected maximum. We

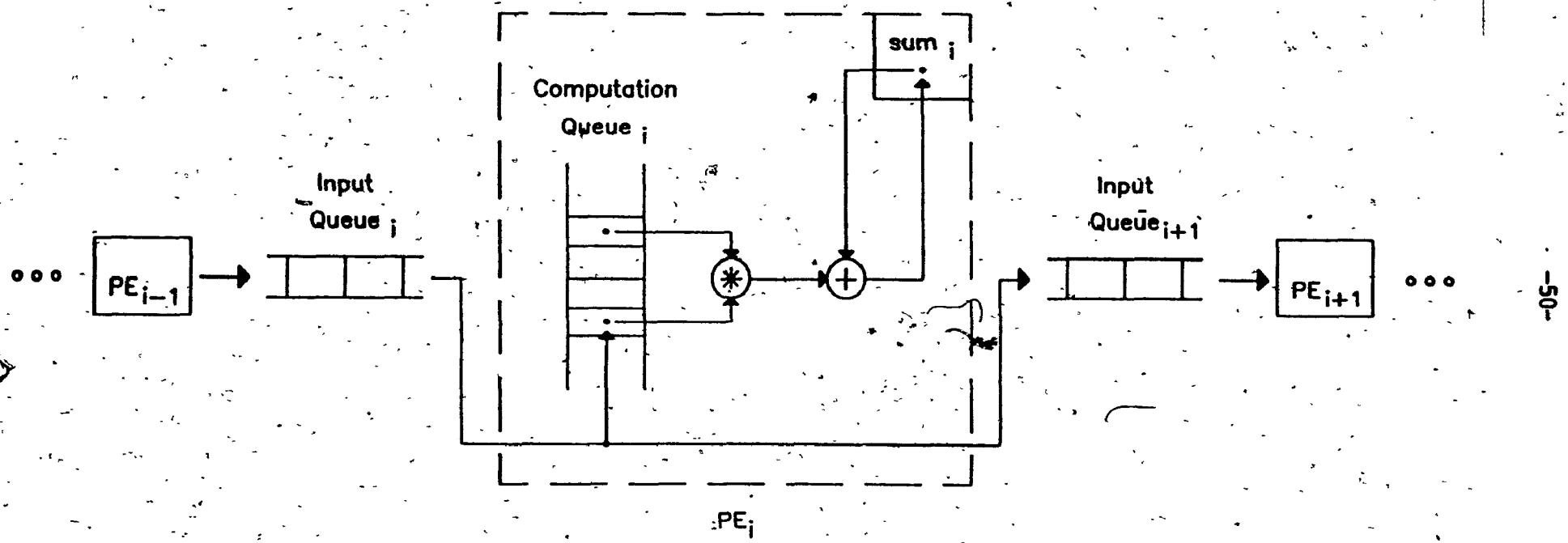


Figure 2.12 Overall Structure of the Autocorrelation Computation in the Homogeneous Multiprocessor.

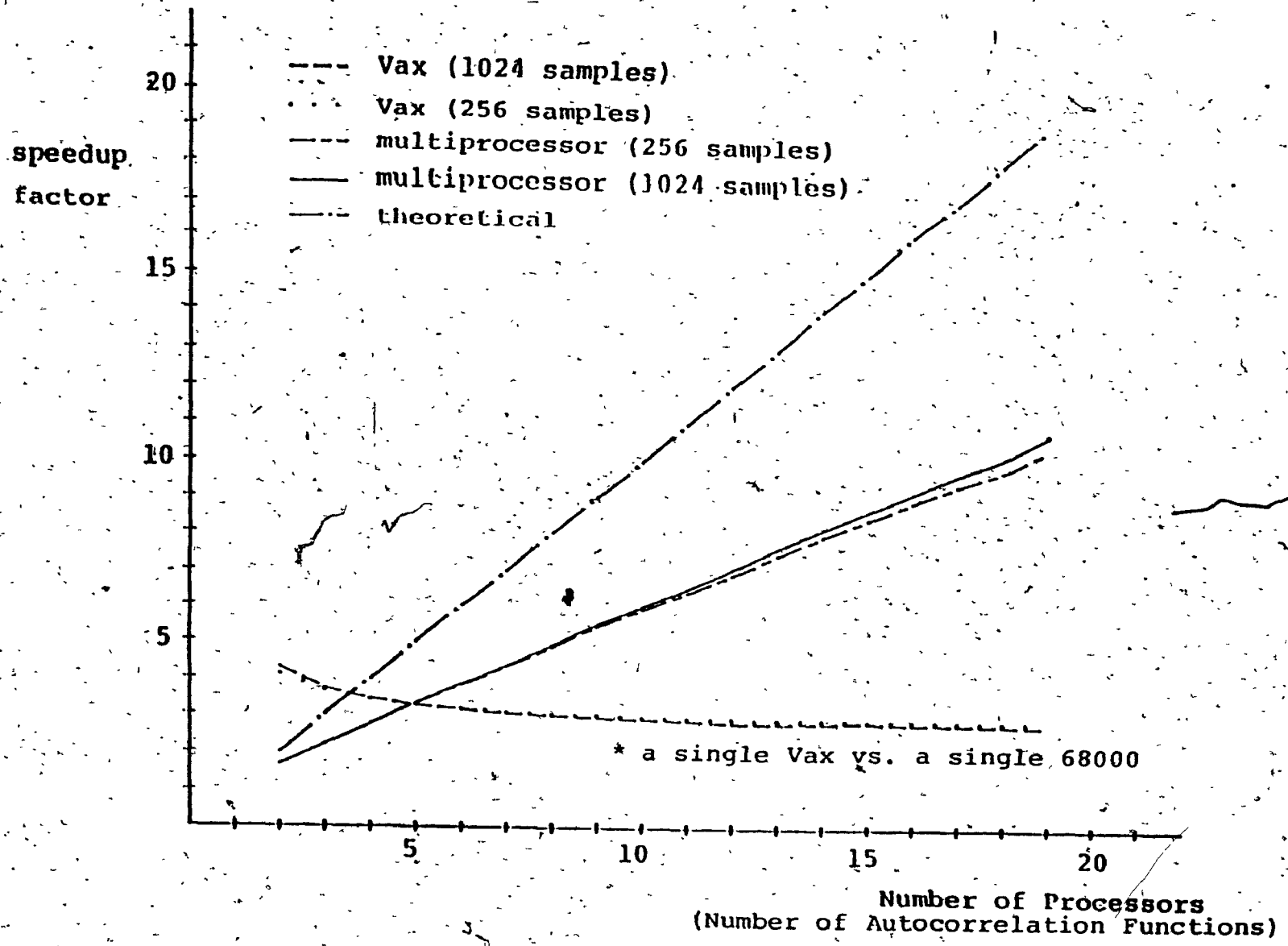


Figure 2.13. Speed-up Factors for the Autocorrelation Computation

have also included, for comparison purposes, the speed-up factor obtained when the algorithm is executed on a single VAX 11/780 machine. Also, incorporated in the overall time of calculation, for the multiprocessor case, is the time needed for the data to propagate to the last processor. It can be observed that the obtained speed-up factors behave almost linearly with the number of processors, yet they are lower than the expected theoretical maximum. This can be attributed to the overhead due to the management of the input and computational queues maintained by each processor as well as the actual movement of data from processor to processor†. Observed is an improvement in the speed-up factor in the case where the number of samples increases, since the time needed for the data to propagate to the last processor is small compared to the overall calculation time.

Nevertheless, in our implementation we were able to obtain a speed-up factor of 10.75 for the case of 19 processors processing 1024 samples, and to surpass the performance obtained on a VAX 11/780 machine with a Homogeneous Multiprocessor composed of five or more processors.

2.5. Discussions and Conclusions

In this chapter, we have presented the model and the structure of the simulator, used as a tool for the performance study of the Homogeneous Multiprocessor processor array. In the implementation of the simulator, every effort and consideration were made to model the actual operation of the Homogeneous Multiprocessor as accurately as possible, though some assumptions have been made based on the projected hardware. These assumptions, however, can be easily modified to reflect any changes in hardware, due to the modular structure of the simulator.

† in a systolic array implementation, the management of such queues is delegated to hardware, and thus the speedup can be further improved.

These changes may include the propagation delay of the bus handshaking signals, the speed of the processor clock, and the preferences in deadlock arbitration and in the switching algorithm. For example, if preference was given to the other deadlock processor instead of the one chosen as described, changes in the simulator will be confined to the procedures describing the switching module. As a matter of fact, the same set of simulation experiments can be performed again with changes in preference in the deadlock cases. Thus, the behaviour of the network can be re-examined with a different decision policy, though similar results are expected.

The results obtained from simulation on the behaviour of the switching network are very encouraging. In most multiprocessor systems, contention for the pathways that lead to common memory are quite often considered to be a major factor on the efficiency of the system. Our system shows good performance within an acceptable bound even under intense load conditions, which is evident by studying the average memory access time, the average idle time, and the average wait time shown in the plots presented.

The Homogeneous Multiprocessor also shows great promise in an application environment. Besides the simulation experiments presented in this chapter, several image processing algorithms have been simulated on the simulator. The simulation result of parallel algorithms for low level vision on the Homogeneous Multiprocessor can be found in [53]. The speed-up factor obtained in running a distributed algorithm for the calculation of autocorrelation functions is a good indication of how the Homogeneous Multiprocessor will perform in executing distributed programs, even though it is conceived as a more general purpose machine for the class of applications involving nearest neighbours processing. It is worthwhile to mention that the simulation of the autocorrelation functions calculation was performed to study the efficiency of the hardware, not the effectiveness of the distributed algorithm simulated. The Homogeneous Multiprocessor is not the best possible hardware for such an algorithm. However, it does provide an insight and acts as a tool to evaluate such an algorithm implemented on other hardware

structure, such as a systolic array architecture. A more realistic approach using the Homogeneous Multiprocessor would have been to broadcast the sampled data over the H-Network to all the processors in the array. Each one will then calculate one auto-correlation function without the overhead of transferring data to its neighbour.

The simulator has given us some valuable ideas of how the Homogeneous Multiprocessor operates. It has also helped in improving certain design policies. For instance, the deadlock cases occurred in the switching network were discovered through initial simulation runs. The simulator can also be used for the testing of distributed applications. The logical structure of an application can be verified on the simulator, thus saving considerable amount of time in the debugging process on the actual hardware.

3. An Operating System Nucleus Design

3.1. Operating System Nucleus Approach

Performance studies for both the processor array and the H-Network [21,42] have yielded excellent results. However, before any actual implementation of applications, an operating system development is necessary to provide users efficient utilisation of the raw hardware.

One of the first operating systems designed for a multiprocessor was Hydra, a kernel for the C.mmp system [69]. The primary goal of Hydra was to permit operating system services and facilities to be implemented as user-level (user-defined) programs on top of the kernel. The designers of Hydra intended to create an environment for effective utilisation of hardware resources, through a capability-based protection mechanism. Another attraction of Hydra is the separation of policy and mechanism. Policies are encoded in user-level software that pass parameters to the kernel where they are implemented. Parameterised policies are thus executed in a protected hardware environment. For instance, Hydra provides three synchronisation mechanisms (besides the user-level spin locks), and two levels of scheduling mechanisms from short to long term periods.

The Medusa operating system was written for the Cm* multiprocessor system [49]. The operating system utilities provided are modeled after Unix's. The major feature of Medusa is its distribution of operating system services. The operating system is partitioned into disjointed utilities including the memory manager, the file system manager, the task manager, the exception reporter, and the debugger/tracer. These utilities, as well as user programs, are composed of task forces which are distributed to individual processors for execution. All processors are capable of executing any activity of a task force. In Medusa, a processor is only allowed to execute code that resides in local memory. This restriction minimises memory access delays and avoids memory conten-

tion. A message-transaction-based interprocessor communication mechanism, using intermediate objects called pipes, is provided for remote function invocation. Another interesting concept concerning the task force execution is co-scheduling. To achieve maximum utilisation of hardware, and avoid unnecessary delays, blockings and communication, a certain number of related activities are scheduled to run simultaneously. This strategy is similar to the one involving the working set of a paging memory system.

Our operating system is based on a nucleus structure — the HM-Nucleus [40,41]. The HM-Nucleus provides the basic mechanisms for the utilisation of the bare hardware, thus enabling resource management and application software to be conveniently built on top of it. It also provides primitives for interprocess communication, capability checking, memory management, process management, and I/O handling. An identical copy of the HM-Nucleus resides in each individual processor.

Having copies of the nucleus residing in each processor is necessitated by the fact that no globally shared memory exists in our system. On the other hand, we envision that operating system utilities will be residing in a few nodes and applications will be assigned to them on demand (as it is done for Medusa) when a complete version of the HM-Nucleus is implemented.

We intend to provide an environment where the execution time of an application is minimised. Hence, an application remains resident until it finishes executing (i.e., no swapping). It might seem that such an approach under-utilises processors; however, in a multiprocessor system, an abundant number of processors exist. This tradeoff is justified in recognising the gain of optimising application execution time.

Since, at this point, we are interested in using the Homogeneous Multiprocessor for compute-bound parallel applications, the initial design for the software thus contains only those portions of an operating system which are sufficient for the execution of long-running applications. We forego the implementation of those functions pertaining to code development, file system, distributed resource management, and so on. Software

code is currently developed in a user-friendly uniprocessor environment provided by a front end running a standard operating system (e.g., Vax-780/4.2 BSD Unix). Program code and system code, generated as a unit, are subsequently loaded into the Homogeneous Multiprocessor for execution.

This approach is similar to the one taken by some other systems. The commercially available Butterfly Parallel Processor uses a host computer for software development, and then downloads the object code to the system for execution. Its operating system, Chrysalis, offers no resource allocation, no process migration, and no file system. A user program must create its own processes and decide where those processes will run [14]. Such an arrangement permits early development of interesting applications.

The structure of the HM-Nucleus itself follows a hierarchical or layered design. This functionally partitioned hierarchy enables information encapsulation and separation of policy and mechanism. Such structure facilitates any modification or addition of modules both within and outside the HM-Nucleus. In addition, concurrent designs of software modules can be easily integrated as long as the visible specification of these modules are well defined. Thus, software development based on a layered structure can potentially speed up the design and implementation procedure.

3.2. Layered Approach in Operating System Design

Our HM-Nucleus implementation follows the model suggested by Brown, Denning, and Tichy [11], according to which the hierarchical structure of a model operating system consists of the following levels, from 1 to 15: electronic circuits, instruction set, procedures, interrupts, primitive processes, local secondary store, virtual memory, capabilities, communications, file system, devices, stream I/O, user processes, directories, and shell. The first eight levels (1-8) correspond to the general single-machine operating system structure, while levels 9 through 14 correspond to the multimachine levels. The last level (level 15) is the user interface, the shell.

Of particular interest are levels 9 through 12, the communications, the file system, the I/O device, and the stream I/O levels. In a hierarchical structure, such as the one described, a single operation is sufficient to manipulate different object types at a particular level. For example, a common 'Open' operation is employed to open pipes, files, and external devices at the stream I/O level. This approach encourages a uniform access to the functions available in a level, and makes the integration of visible specifications from different layers easier.

This layered, hierarchical structure, which provides the template between hardware and software, fits nicely into our requirements and intentions in designing the operating system for the Homogeneous Multiprocessor. Several layers, especially communications, map directly to our hardware. In our design, however, we deviate from the hierarchy suggested by Brown et al. Since the Homogeneous Multiprocessor does not provide a local disk at every node, and disk space is distributed among a limited number of specialised nodes, the H-Network is used to transfer information between the disk and the node requesting it. Therefore, we place the virtual memory above the communications layer. In addition, both the H-Network and the extended buses are used to provide interprocess communication. Thus the communications layer is placed much closer to the bare hardware than in the model suggested by Brown et al.

Another noticeable difference between the hierarchically dispersed functions within the HM-Nucleus and some other layered structures such as the International Standard Organisation's (ISO) Open System Interconnection (OSI) [16] is that in the HM-Nucleus, commands from a higher layer do not necessarily pass through every single layer below it. Any service performed in the lower levels can be directly requested by an upper level, bypassing the layers in between. This has the advantage of saving procedural time in requesting services by higher layers in the system. However, the OSI standard is intended for the integration of heterogeneous systems and such a restriction will guarantee the compatible interfacing among these systems.

3.3. The HM-Nucleus

The overall conceptual structure of the HM-Nucleus is presented in this chapter. As shown in Figure 3.1, the HM-Nucleus consists of eleven layers. From bottom up, the layers are Kernel, Physical Memory Management, Device Management, Capabilities, Universal Datagram Services, Remote Procedure Call, Communications, Virtual Memory Management, File Management, Table, and User Interface.

A Homogeneous Multiprocessor node includes the Main Processing Unit (MPU) and its associated memory module, the Memory Management Unit (MMU), the interbus switches, and the H-Network interface. The Kernel is the hardware/software interface layer that provides mechanisms to drive the hardware within a single node, and it incorporates no policy-making modules. These Kernel primitives serve as extensions of the bare hardware and are used by higher layers. These extensions include process switching, primitive I/O, interrupt handling, and MMU manipulation.

The Physical Memory Management layer is responsible for the allocation and deallocation of memory space for processes and communication packets, and, with the cooperation of the MMU, provides virtual-to-physical memory mapping and low-level access rights checking.

The Device Management layer provides device drivers for the peripherals attached to a node. Usually the only devices will be the H-Network controller and the channels for neighbouring-processor communications, but specialised nodes may include disk controllers.

The bottom fourth layer, or Capabilities, provides mechanisms for the creation and management of capabilities. A capability, in the context of the HM-Nucleus, is a unique name consisting of a type, a processor number, and an identification that points to the information concerning that object.

The Universal Datagram Service (UDS) layer implements various interprocessor

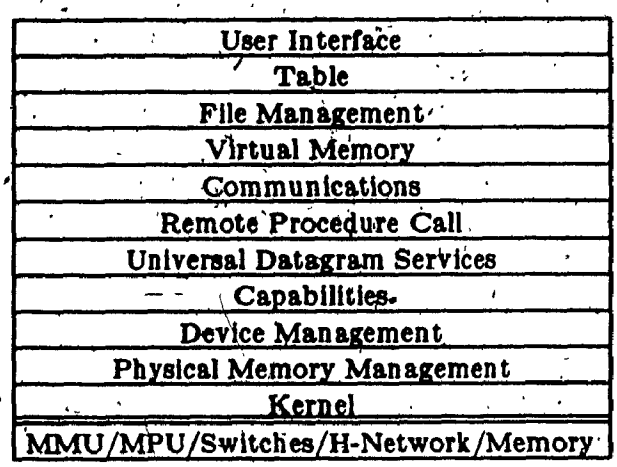


Figure 3.1 The Layered Structure of the HM-Nucleus

communication protocols. The placement of the UDS layer that low in the hierarchy is necessitated by the capability of our architecture to function as a multiprocessor in addition to being a distributed system. This is also an important difference between our model and the one proposed by Brown et al [11].

These five layers, from Kernel to UDS, roughly correspond to the single machine levels (1-8) of Brown's model. The Kernel provides the functions of levels 1 to 5 while the Physical Memory layer manages physical memory. The remaining layers of our model, on top of these five levels, can be viewed as multimachine levels.

On top of UDS layer is the Remote Procedure Call (RPC) layer. This layer serves as an interface between the abstractions of single machine and multimachine. Any inter-nucleus communication initiated by a local machine is routed to the appropriate recipient, through the UDS by the RPC Manager.

The next layer in the HM-Nucleus is the Communications layer. This provides communication facilities between *user* processes (processes outside the HM-Nucleus), in the form of pipes and channels. This layer is based on the UDS layer discussed above, and use the facilities of the H-Network for process-to-process or broadcast-mode communications, and the extended bus for neighbouring-processor communications.

The Virtual Memory Management (VMM) layer is responsible for assigning and managing virtual space for processes in collaboration with the Physical Memory Management layer. There are three kinds of such space: program, data, and stack space for user programs; shared regions; and global memory. At a later stage of our development, swapping of user spaces will be implemented in this layer.

The File Management layer will implement a hierarchical file system, modeled after the Unix file structure [56]. Branches of the tree will be situated at nodes possessing a local disk, while nodes without disks will implement the semantics of the Unix file system interface and refer all requests to a server node. Server nodes will have a complete

file system manager and will store portions of the file system data on their local disks. The initial implementation will support only access to the device (special) files.

The Table layer at the present time simply consists of catalogues where the correspondences between symbolic names and their capabilities are entered. These catalogues are currently resident in the machines to which the capabilities belong. It is intended that this level will be further developed into a distributed directory layer at a later stage.

The outermost layer is the User Interface layer. Initially, this will contain the actual code of the application. Subsequent versions will be capable of managing user processes and responding to a general system call interface, thus providing the minimum user interface requirements to run application programs.

We are using Concurrent Euclid (CE) [33], a concurrent language, as a programming tool for the implementation of the HM-Nucleus and user applications. The research team at University of Toronto has implemented a Unix-compatible nucleus called Tunis, which is written in CE. Tunis has a layered structure with the following levels, from the bottom: Kernel, Utility, Device, Memory, File, and User. Codes for Tunis have been written for the PDP-11 and the NS32016. A standalone kernel has also been written for the MC68000. Since this bottom layer of the Tunis operating system provides the basic mechanisms for hardware utilisation and supports CE constructs such as processes and monitors, the Kernel layer of the HM-Nucleus is therefore based on the Tunis Kernel. However, we have expanded and modified it to our specific needs. A description of the original Tunis Kernel, extracted from the documentation which accompanied the Tunis operating system code, can be found in Appendix G.

The MC68000 hardware provides two operating states: the user and the supervisor. Since the HM-Nucleus primitives are protected entities, they are executed in the supervisor mode and protected by mechanisms provided in the MMU. User programs, on the other hand, are executed in the user mode. Switching of execution environment through

system calls available in the User Interface layer enables a user process to request system services. System envelope processes waiting for user system traps at the User Interface layer will execute system code on users' behalf.

Our current interest, though, is to verify the system code designed and to run applications programs immediately. Therefore, the HM-Nucleus code and the application code are compiled, linked, and downloaded to a node as one unit. Since an initialisation program that delegates system and user space has not been implemented yet (e.g., system heap and user heap allocation), this piece of code is executed in the supervisor mode to satisfy our current needs. However, when a complete version of the HM-Nucleus is implemented, applications programs will be running in the user mode on the hardware.

In the next few sections, the specifics of the structure of the various levels of the hierarchy are given, and visible functions are presented using the notation of Concurrent Euclid [33], starting with the Kernel. Detailed specifications of all these functions and procedures available in various layers are presented in Appendix H.

3.3.1. The Kernel

The lowest level of software/hardware interface is implemented in the Kernel layer. This layer provides mechanisms to drive the hardware and no policy making module is implemented, which allows us to have flexibility and efficiency in designing system software. The Kernel module provides extensions of the bare hardware, which are used by the higher layers (operating system and applications). These extensions include process switching, primitive I/O, interrupt handlers, and MMU manipulators.

The Tunis Kernel has provisions for enabling and disabling external interrupts coming into the local node. Since the Kernel is the lowest module of the HM-Nucleus residing in individual processors and does not interact directly with the rest of the system (interprocessor communication is handled by higher layer software), it thus acts as a

single entity and hence single-node mutual exclusion can be enforced. This fact is reflected in the use of monitors [31] as a synchronisation tool within the HM-Nucleus, by using the facilities available in the Kernel.

The Tunis Kernel also provides synchronous I/O primitives for the Device layer so that I/O startups, acknowledgements, and responses are hidden within it. Transparent to the users, the I/O handlers are perceived as operations in synchronous mode, with the device invokers waiting in queues for the completion of requested operations. Thus, programmers are relieved from the complex procedures associated with asynchronous I/O operations.

We have made modifications and additions to the Tunis Kernel to meet our specific needs. The seven prioritised hardware interrupts described in section 2.1 are intercepted by interrupt handlers in the Kernel. These interrupt and trap handlers carry out preliminary processing and route to the proper managers or the device drivers at higher levels upon receipt of an exception.

Operations involving the MMU, such as those of loading a MMU descriptor and writing to a MMU register, are implemented in the Kernel layer. These operations, though, leave any decision making to the Physical Memory Management and the layers above it.

To load a descriptor into the MMU, we use the Kernel function *LoadDescriptor:*

descriptor := LoadDescriptor (vir_mem,status)

where *vir_mem* is a pointer used to retrieve information about the segment such as address range, starting virtual address and starting physical address, as determined by the *BindSegment* call (provided by the Physical Memory Management layer). This information is organised in accordance with the Motorola MMU descriptor format. The *status* parameter indicates whether the operation was successful or not. In all the remaining HM-Nucleus system calls, we use the variable *status* to return the status of

the call. The returned parameter *descriptor* indicates the index of the descriptor in the MMU.

To obtain information on a particular descriptor in the MMU, the Kernel procedure call *ReadDescriptor* is used:

ReadDescriptor (info,descriptor;status)

Here *descriptor* is the descriptor desired to be read (out of the 32 possible descriptors in the MMU); its contents are stored at the location indicated by *info*. These two Kernel routines are very useful in memory segment management in the processes of allocation and deallocation of physical memory.

To aid the synchronisation of referencing common memory space (for details refer to section 4.2.1.1 on shared regions), and the protection of unsharable information, the following two calls are issued with *descriptor* being the MMU descriptor number:

EnableDescriptor (descriptor,status)

DisableDescriptor (descriptor,status)

When an MMU descriptor is disabled, its specified virtual-to-physical address mapping will not be provided. There are also some auxiliary routines (see Appendix H) encapsulated in the Kernel layer which are not visible to other layers, but which help the efficient implementation of the visible ones.

3.3.2. Physical Memory Management

As mentioned earlier, each processing element consists of an MC68000 plus an MC68451 Memory Management Unit, which provides virtual address translation. When a virtual address is presented, the MMU translates this address according to the matching descriptor and puts the actual physical address on the address bus. Details of the

MMU address translation algorithm are presented in Appendix I. This hardware provides a 16 Mbyte virtual address space, which we plan to map to physical memory as follows: 0—4Mbytes : local memory; 4—8Mbytes : left memory module; 8—12Mbytes : right memory module; and 12—16Mbytes : I/O devices. The present implementation provides a 1M byte local memory module plus two extra 1M byte nonlocal memory modules belonging to the neighbours on the left and the right. This nonlocal memory is allocated by the Physical Memory Manager in the form of regions and is used by the shared region management facilities in the Virtual Memory Management layer.

For processes running on the local processor, higher-layer managers issue memory space requests to the Physical Memory Manager. Such requests specify the virtual address range, and it is the responsibility of the Physical Memory Manager to allocate a physical memory address space. If no single hole existing in physical memory is large enough to accommodate the request, the physical memory manager executes a modified version of the Buddy algorithm † to find enough holes to satisfy the request. The physical memory space allocated is outside HM-Nucleus code and data spaces in the memory. Since applications and HM-Nucleus execute as one unit in the current implementation, the allocated spaces are accessed under supervisor data mode. In the future, such allocated spaces will be assigned as either supervisor or user data segments.

The Buddy algorithm we use is tailored to the MMU used. Certain hardware restrictions are imposed: (1) the size of a block mapped by any single descriptor must be a power of two with a minimum size of 256 bytes and a maximum size of 16M bytes; and (2) there are only 32 descriptors per MMU. On the other hand, a segment can be

†The Buddy algorithm used for dynamic storage allocation is described by Knuth [35]. This algorithm allocates memory in blocks of 2^k bytes where k is some integer less than the maximum m (for a total available memory of 2^m). Separate lists of available blocks, each of size 2^k, $0 <= k <= m$, are kept. When a block of 2^k bytes is requested and none is found in the 2^k block list, a larger block, 2^{k+1} or larger, is split into two equal halves called buddies. This process repeats until a block of 2^k bytes is obtained. A reversed process will merge two free buddies and put the resulting block back into the corresponding available list.

mapped onto several, not necessarily contiguous, blocks by using more than one descriptor. Taking into account the above requirements, we impose another restriction in the HM-Nucleus: all segments assigned must have sizes that are multiples of a minimum block size (for example, 8K bytes). Our current implementation, however, only allows contiguous blocks allocation. The Buddy algorithm is then used to allocate the necessary number of physical memory blocks by using an optimum number of descriptors.

For both local and nonlocal (shared region) memory allocation, the Physical Memory Manager uses the functions available in the Kernel to load the virtual address and the corresponding physical address range into the appropriate descriptor(s) of the MMU. At the same time, the write-protection bit in the segment status register of the MMU descriptor is set according to the access mode indicated by the request. This facility enables low-level enforcement of access rights, since the MMU will generate a fault when a write-protected segment is accessed in the write mode. Also, the enable bit in the segment status register could prohibit any access to the segment at all, if desired. This mechanism facilitates the implementation of critical regions (as is discussed in section 4.2.1 on the implementation of shared regions).

The Physical Memory Manager manages the physical memory space through various data structures existing in this layer. A physical memory table, consisting of a list of segment descriptors, holds all the information about the system physical space. As shown in Figure 3.2, a segment descriptor corresponding to an allocated segment consists of four fields: a segment descriptor number; a pointer to the physical memory descriptor which indicates the task number, the first, and the last block numbers; a pointer to the virtual memory descriptor, which holds information on the virtual-to-physical address mapping and the access mode of the segment, to be used as a template for loading the MMU descriptor; and an active field to indicate if a MMU descriptor has been loaded with the information or not. There are several free lists in existence: free lists for available segment, virtual memory, and physical memory descriptors; and a free list consisting of available physical spaces in different block sizes.

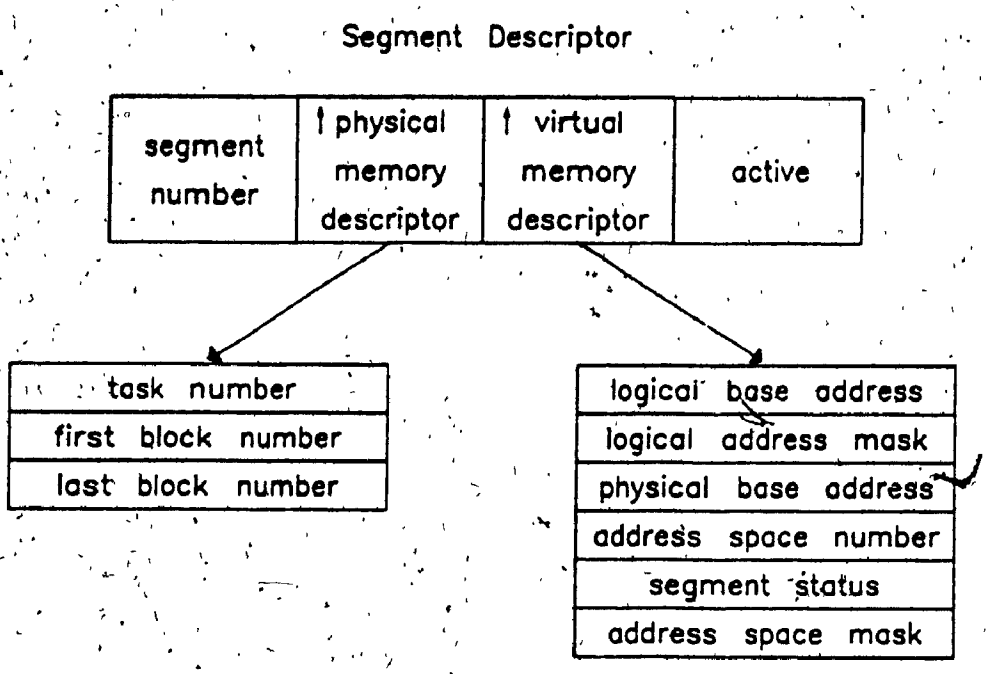


Figure 3.2 Segment Descriptor

There are four function calls provided by the Physical Memory Manager:

seg_num := AssignSegments (size,task,status)

The *AssignSegments* function, as discussed earlier, uses a modified Buddy algorithm to assign the appropriate physical memory blocks. The parameter *size* is the size requested and *task* is the task number of the invoker. The returned parameter *seg_num* is the number of the segment descriptor assigned.

vir_mem := BindSegment (seg_num,task,port,access,status)

The *Bindsegment* call creates the virtual memory descriptor; *task* is the task number; and *port* is the starting virtual address of the segment passed along with *access* (which indicates the access mode of the segment) by the invoker. The returned parameter *vir_mem* points to the virtual memory descriptor. Also, all the information pertinent to the segment is entered in a segment descriptor (specified by *seg_num*) of the physical memory table for memory space management purposes.

When a segment is no longer needed, the higher layer managers issue the *DeleteSegment* call:

DeleteSegment (seg_num,status)

The corresponding active entry of *seg_num* in the physical memory table is marked as inactive to indicate that the deallocated space can be allocated to someone else. The corresponding segment is returned to the free list, and the MMU descriptor is disabled.

Another function of the physical memory manager is garbage collection, which is invoked by *MakeSpace* upon unsuccessful termination of the *AssignSegments* call. The garbage collector examines the segments that are no longer needed (either the assigned process has been destroyed or the packet used for communications has been consumed), and enters any newly created free blocks into a free list for future allocation. Algorithms

in the *MakeSpace* procedure also try to make segments in different sizes available.

3.3.3. Device Management

The Device Management layer provides software/hardware interfacing with device drivers to individual physical devices that are attached to a node. The communication connections between neighbouring processors utilising the extended bus mechanism are the channels. Communication packets are assembled by the Universal Datagram Services layer and their delivery and handshaking are the responsibility of the channel drivers. Details of the communication packets and their delivery mechanisms will be presented in section 4.1.1. Besides the channels to the left and the right neighbours, a necessary driver for each node is the H-Network controller driver, since all the nodes in the Homogeneous Multiprocessor are connected to the network. In addition, if a node is configured as a disk server, a disk driver will be resident in that node.

Two RS-232 drivers have also been implemented. One of these can be used for the purpose of monitoring the system and for operator interaction if one of the processors is configured as a front end. It is also very useful in the process of verifying and debugging hardware nodes and software development modules.

The second driver is currently used for communications with the host computer to edit, compile, and download executable programs. It can also be used by a back end processor to communicate with a host computer.

3.3.4. Capabilities

This layer provides a naming mechanism for the rest of the HM-Nucleus. The naming mechanism follows the capability convention to create a unique name. Since the hardware assist needed for capability checking is not part of the processor used, we forego the implementation of object protection mechanisms [3]. Such mechanisms may appear in a future implementation. Presently, this layer has a single visible routine

CreateName, used for unique capability generation:

$$cap := CreateName\ (type, id)$$

cap is in the form of *type:processor:id*, where *type* denotes the type of object, *processor* the owner of the object, and *id* points to a collection of descriptors which contain information concerning that particular object.

Mapping and searching of capabilities are the responsibility of the Table layer. More details on the discussion of capabilities are presented in section 4.3 (on global identifiers).

3.3.5. UDS, RPC and Communications

The Universal Datagram Services (UDS) layer provides a uniform access interface to the underlying hardware communications channels. These channels include the H-Network, the extended buses, the RS-232 serial lines, and so on. The unit of transmission is a datagram of arbitrary size, which permits specialised protocols to be used among the (HM-Nucleus) processes that are responsible for the management of distributed services. Possible protocols to be supported include the IEEE 802 standard for Local Area Networks [4], the X.25 [1] and the Transmission Control Protocol/Internet Protocol (TCP/IP) [52]. The UDS formats the appropriate datagram for the specified communications protocol and invokes the corresponding driver.

The UDS layer currently supports packets delivered through the extended bus mechanism using virtual channels available in the Device layer to either of its two immediate neighbours. Two types of packets are currently under implementation: a normal data packet that encapsulates data and a special control packet that forces the recipient processor to interrupt its neighbour located opposite to the sender. Details of communications packets and their delivery mechanisms are presented in section 4.1.

The UDS layer provides mechanisms for communication purposes and its facilities can be used by both the Remote Procedure Call and the Communications layers. These two upper layers implement the semantics and policies for interprocessor and interprocess communications. The UDS layer, therefore, will be implemented with enough fundamental functions so that policy changes will affect the Remote Procedure Call and the Communications layers only. The UDS layer can also be expanded to encompass any additional communication pathways deemed necessary in the future (e.g., an interface to ARPANET), as a common protocol adapter for internetwork communication [50].

The Remote Procedure Call (RPC) layer, as its name implies, allows user processes on a local node to perform a remote procedure call to servers distributed in the system. It removes from the user process any responsibility in communication handshaking and protocol overheads. Any nonlocal call would appear as a local call with the only noticeable difference being the delayed response as compared to the case of a local call. This layer operates independently of the Communications layer.

The Communications layer provides interprocessor communication primitives in the form of pipes through the H-Network, and channels and a remote signaling mechanism with neighbouring processors through the interbus switches. Pipe communication facilities utilising the H-Network form a basis for a message passing system. However, implementation details will not be given here since the hardware specification for the H-Network has not been finalised yet. The communication connection for neighbouring processors, utilising the extended bus, in the form of channels established during system start-up, and the remote signaling mechanism, using device drivers in the Device layer, are presented in detail in section 4.1.

These three layers, UDS, RPC, and Communications, handle all the interprocess and interprocessor communications for a local node in the Homogeneous Multiprocessor. The RPC and Communications are two independent entities and both utilise the facilities available in the UDS to carry out their requested services. The placement of these

two levels above the UDS provides us the capability to study the approaches to communications in a distributive environment: the remote procedure call approach and a message based system, pertaining to the hardware available in the Homogeneous Multiprocessor.

3.3.6. Virtual Memory Mangement

Responsible for managing the virtual memory space for a local node, the Virtual Memory Management layer implements the following virtual spaces: (i) program, data, and stack areas similar to Unix's for local users; (ii) shared memory among three neighbouring processors in the form of shared regions; and (iii) global memory in the system.

For local user programs, the Virtual Memory Manager cooperates with the Physical Memory Manager, allocates available memory segments, and provides virtual-to-physical address mapping. Presently, the functionality available for this purpose is is minimal because our immediate goal is to run application programs which are resident in the memory until they finish. Therefore, the virtual space is rather easy to manage. However, more sophisticated facilities, including remote swapping, will be provided in a more general purpose system in the future.

The second type of virtual space to be managed is shared regions. In the context of the Homogeneous Multiprocessor, a shared region is a data structure that can be *potentially* shared by three processes running on three adjacent processors. The Shared Region Manager is entrusted with the creation of shared regions, the binding of a shared region to a specific port in the virtual space of a process, and finally safeguarding the entry to the region.

Procedures are provided to create, bind, and delete these regions. A detailed explanation of how shared region management is implemented can be found in section 4.2.1, where the issue of mutual exclusion in entering the shared region is also discussed. In the following, we present the syntax for the available calls:

reg_cap := CreateRegion (size,guarded,status)

a shared region is created with the given *size*, and a region capability *reg_cap*, pointing to a collection of descriptors for the shared segment, is returned. The Boolean *guarded* indicates whether the segment will be shared on a mutually exclusive basis.

lock_cap := BindRegion (reg_cap,port,status)

The *BindRegion* call binds a region created to a specific *port* in the virtual address space of the calling process; a *lock_cap* is returned which points to the location of the locks used by the mutual exclusion mechanism.

EnterRegion (lock_cap,status)

EnterRegion is used to obtain the mutually exclusive right to access the shared region, while *ExitRegion* gives up that right:

ExitRegion (lock_cap,status)

UnBindRegion (reg_cap,port,status)

DestroyRegion (reg_cap,status)

UnBindRegion disables the calling process from accessing the shared region while *DestroyRegion* is used to deallocate the memory space occupied when the shared region is no longer needed.

This layer is also responsible for the management of global data structures. Mechanisms will be provided (at lower levels) for the policies implemented. With the intended implementation of global data structures, data consistency and updating methodologies are devised. Details of our approach to these problems are presented in section 4.2.2.

In our future plan, we intend to support secondary storage by designating nodes in our system as disk servers. This layer will therefore be responsible for swapping operations. This will involve a sleeping swapping process to be awaken by the Physical Memory Manager when no more physical memory space is available in the local node. This awakened process will proceed to do swapping at specialised nodes in the system based on the information supplied by the Virtual Memory Manager, together with the cooperation of the File Manager.

It is our view that this layer should consist entirely of policy making modules available to users, while mechanisms to support these policies are provided by lower levels. For example, visible procedures available to users to implement relocation, placement, and replacement strategies concerning memory management, are parameterised. This will facilitate the addition and modification of decision making modules as they are confined in this layer. Also, parameterised decision modules can help us study the efficiency of various management algorithms.

3.3.7. File Management and Table

The File Management layer deals with operations on named files (local and remote) found in the Table layer. Similar to the UDS, RPC, and Communications layers, facilities in this layer have not been fully defined yet; however, we intend to implement functions in a stream device fashion such that the file structure and its operations will be compatible to Unix's.

The Table layer currently consists of catalogues where correspondences between symbolic names and their capabilities are entered. The Table Manager is also responsible for system name binding where a user symbolic name is bound to the corresponding machine capability (located either locally or remotely). This layer is intended to be further developed into a distributed directory layer.

Exported calls in this layer include *GetName*, *Map*, and *UnMap*. *GetName* is used to get the capability corresponding to a user symbolic name, even when some of the parameters are missing, such as the case of unspecified processor:

cap_list := GetName (name,task,processor)

where cap_list is the returned list of capabilities (normally a single item list), associated with the symbolic *name* (a string of ASCII characters of arbitrary length); *task* is the capability of the task under which the capability of the *named* object was created; and *processor* is the processor on which the *named* object was created.

The *Map* call maps an object with a capability, while *UnMap* deletes the object from the map table:

Map (cap,name,access)

UnMap (cap,name)

name is the user symbolic name; *cap* its capability; and *access* the specified access right.

The effects and specifications of these visible functions may be better understood after reading the discussion on the general design philosophy of capabilities, which is presented in section 4.3.2.

3.3.8. User Interface

There are two types of runnable processes: system and user. System process runs in the domain of the operating system and its execution is under the supervisor state. User processes represent user programs and run in the less privileged user state.

Envelope system processes are guardians for user processes and execute system code on behalf of the user processes [33]. The User Interface layer has several envelope system processes waiting at system traps so that user processes are provided the ability to make system calls in the system domain. These system processes take care of local user

processes' system calls as well as remote procedure calls originated by processes (either system or user) residing in processors other than the local one. These remote processors request services either through the extended bus, if it is a neighbouring processor or through the H-Network, if it is a distant processor.

The envelope system processes are invoked every time a software system trap is encountered. In the current implementation, the line 1111 Emulator trap is used as the system call trap. However, in a future implementation, a TRAP N instruction will be used instead. The TRAP N instruction provide up to 16 different categories of system calls and these calls can be relocated without having to recompile the applications. The line 1111 Emulator trap is realised if the most significant four bits of an instruction word being 1111 is decoded. By interpreting the call number encapsulated in the instruction word, a system request is routed to the appropriate manager. This system call implementation gives the user processes the power to execute system programs through a change of execution environment, i.e., switching the processor from user operating mode to supervisor mode. An envelope system process also handles user process management and is capable of spawning, killing, suspending, and resuming processes.

Currently, this layer is transparent to the users. System code and applications code are compiled and downloaded as one unit to enable the convenient testing of both system and user programs. When fully developed, user interface specification will be based on the Unix standard [6].

3.4. Discussion

As stated earlier, the objective in designing an operating system for the Homogeneous Multiprocessor is to make our system extensible. By implementing the basic operating system functions in the form of a nucleus, modularity, and decision and information encapsulation are made possible. An additional advantage of this layered, modular approach is the separation of policy and mechanism. With parameterised policy-making

modules available to the user, a very flexible application environment is created.

Another objective in designing the operating system for the Homogeneous Multiprocessor is to "bring" the prominent architectural features of the system to the user level in the form of "secure" abstractions. This approach (1) enables the user to tailor his application so that it closely matches the underlying hardware, and (2) avoids a lot of software overhead. The overall result is code that runs efficiently. We believe that we have struck an reasonable balance between the needs of abstraction and information and decision encapsulation, and the need of an application to execute in the shortest possible time.

Not all the functions available in each layer are defined. For example, the UDS, RPC, and Communications are not fully specified. This is due to the fact that the related hardware components and protocol designs have not been completed yet. However, we have provided enough foundations for the testing of remote procedure call and the message passing facilities, based on datagrams and distributed pipes. For instance, research concerning remote procedure call semantics, such as exactly-once and at-least-once [47], can be carried out using the functionality available.

Similarly, the File Management layer and its internal structure are not defined, but it will most likely conform to Unix's file structure. Since Unix has been emerging as a de facto operating system standard, it is to our advantage to make our system Unix compatible. The specification to the User Interface layer will also be Unix like, as researchers outside our project team are encouraged to experiment with our system.

This chapter has presented the conceptual design of an operating system nucleus for the Homogeneous Multiprocessor. The lowest two layers of the HM-Nucleus — the Kernel and the Physical Memory Management, have been implemented according to specifications. We are in the process of implementing the other layers of the nucleus based on the concepts and specifications described.

We believe that the HM-Nucleus provides enough functionality and support to permit distributed application programming. This support includes interprocessor communication mechanisms, distributed virtual memory management, and global identifiers, and is presented in detail in chapter 4.

4. Software Support in a Distributed Environment

4.1. Interprocessor Communications Mechanisms

The foremost task to be supported by the operating system in a distributed environment is interprocessor communications. Without effective communication techniques, the power of a multiprocessor is drastically diminished. The Homogeneous Multiprocessor configures both as a multiprocessor and as a local area network, therefore means to accommodate these different communication pathways have to be implemented. Utilising the extended bus in the processor array, each processor can communicate with its two immediate neighbours. On the global level, each node can communicate with any other node in the system through the H-Network, either on a one-to-one, or one-to-many basis. This global communication mechanism will be implemented as pipes and is currently in the design stage, since the hardware specification for the H-Network has not been finalised yet. Therefore design and implementation details for the pipe communication facility will not be given here. However, we envision that the network communication protocol in the UDS layer will adhere to one of the standards, such as the IEEE 802 standard for Local Area Networks — Carrier Sense Multiple Access with Collision Detection [4], due to the resemblance of the H-Network structure to that of the Ethernet. This section therefore will describe specifically the communication mechanism between neighbouring processors, and also a remote signaling mechanism for a processor to another that is situated two nodes away.

The communication connection for neighbouring processors, utilising the extended bus as a communication pathway, is the *channel*. Channels are predefined well-known addresses established during system start-up time that are used for handshaking dialogue between adjacent processors. They have fixed locations and are available throughout the system up-time. One might consider the shared regions, which provide three neigh-

-bouring processors the capability to share a common information entity (either data or program), to be described in section 4.2.1, as a communication connection. The shared regions are temporary structures and these connections may only exist for a short period of time. In other words, the channels can be interpreted as permanent interprocessor communication connections while shared regions are the temporary connections for interprocess communications. The reason for this distinction of communication connections is that the system requires initial connections in order to arrange for run-time communications paths. For example, to set up a shared region, all three processors communicate through the channels to complete the handshaking protocol.

4.1.1. Communications Packets and Delivery Mechanisms

The basic unit of information used for interprocessor communications is the packet, or datagram, which is assembled by the UDS layer to implement a uniform communications interface. In this section, we are concerned only with the mechanism to deliver a packet and its acknowledgement as implemented in the Device layer. The exact definition and format of a packet or datagram is handled by the UDS layer. In general, a packet consists of a size field, a header field, and a data field. The size field indicates the total number of bytes within the packet, although network packets may conform to the network communication protocol used and be of a fixed length. The header field incorporates information on the sender's and/or the receiver's address, status of the packet, and the packet type. The data field contains data information to be transmitted. In certain cases, however, the data field might be null if the packet is of a control nature.

A communication packet is delivered through the channel mechanism. A channel is defined as the exact physical address in the system memory space that is accessible by a neighbouring processor. For each processor, there exists the right channel and the left channel in the local memory, and they are used for transmitting the packet pointer to

the neighbouring processors. The locations of these channels are predetermined and their contents are initialised to zero at system start-up. Virtual channels, on the other hand, are the distinct channel numbers used by communicating processes residing on neighbouring processors for identification purposes. For example, two communicating processes will use the same virtual channel number for identification even though all the different virtual channels use the same physical channel.

Packets are stored in the local memory space where a pool of circular packet-buffers is maintained by the Communications Manager. The packet-buffers locations are known to the neighbours so that they can enable the corresponding MMU descriptors matching these addresses. When a packet is created, its address is transformed into a pointer and is subsequently stored in the appropriate channel for the receiving processor. The packet might contain data or control information, however, the delivery mechanism is the same for both.

In general, there are two techniques to signal a processor: polling and interrupt. Software polling is quite often used by a waiting processor for solicited events. We have opted for the interrupt technique because polling tends to waste processing time and most of the signalings in a distributed system are unsolicited. Interrupts are accomplished through the hardware interrupt available on the MC68000. Whenever the line IRQLFT or IRQRHT is asserted, a request has been generated by one of the neighbouring processors and the local processor is expected to respond to this signal.

The process of packet delivery to a neighbouring processor is described in the following steps:

1. The Universal Datagram Services layer assembles the packet according to the protocol used and stores it in the packet-buffer.
2. The channel driver in the Device layer puts the pointer pointing to the buffer location and the virtual channel number into the appropriate channel (left or right

neighbour channel).

3. The channel driver interrupts the receiving processor.

4. The running process at the receiving processor is suspended and the receiving channel driver reads from the channel.

5. The receiving channel driver transfers the packet pointer to a waiting process or a server process according to the virtual channel number, and clears the channel.

6. The suspended process resumes running and the waiting process or server process reads the packet when it is scheduled to run on the receiving processor.

The sending processor cannot send another packet to the same neighbour until the channel has been cleared by the receiving processor. This guarantees the receiving channel driver will read the correct packet pointer and virtual channel number. It is the responsibility of the recipient process at the receiving processor to either copy the packet into its own memory or read the information directly from the neighbouring memory.

The packet remains in the memory of the sending processor until it is read by the receiving processor. Since the circular packet-buffer space is limited, some unread packets might be overwritten prematurely due to the inactivity of the consuming processes. Therefore, associated with each packet is a 'read' flag, which will be set by the consuming process once it has read the packet. This implementation facilitates garbage collection which returns released space occupied by used packets to a free list of packet-buffers. The memory space occupied by the packet-buffers belongs to the HM-Nucleus and not the packet-sending process. Consequently, even if the sender deletes itself before the recipient is ready to read the packet, the information is not destroyed.

This approach avoids the overhead of having the receiving process explicitly send an acknowledgement that it has read the message, to the sending process. On the other hand, it is not very efficient in cases of large messages because the sending process has to copy the message to the buffer and then the receiving process may have to copy the

same message to its memory. However, the channels are primarily used to transfer small messages such as the ones in establishing the shared regions. More extensive and frequent interprocess communication between processes running on neighbouring processors can be achieved efficiently through the shared region scheme.

The process of packet delivery thus involves two phases. During the first phase, the packet pointer and its handshaking is the responsibility of the interrupt handling modules. The handshaking and interpretation of the packet itself is handled by both the consuming process and the producing process in the second phase.

4.1.2. Remote Signaling Mechanism

We have discussed how communication is achieved between neighbouring processors in the previous section. There are some other cases that communication is involved between processors that are not adjacent to each other. The following example represents such a case where communication is achieved between non-adjacent processors using the extended bus. Quite often a processor would like to communicate with a processor two positions away in the array (i.e., P_i to P_{i-2} or P_{i+2}). There are two ways to achieve such communications: either through the H-Network or by the extended bus. If it is a long message containing data or complex control information, the H-Network is used due to the efficiency of pipes. Even though we have not made a formal study of the efficiency of the two communication mechanisms, it is very likely that if the message is a simple control instruction, the route by the H-Network is more expensive due to high start-up and network access time. In this case, a processor can request the intervening processor (i.e., P_{i+1} or P_{i-1}) to perform a simple operation, for example, the clearing of a flag. A specific usage of such a mechanism can be found in the implementation of spin-locks for the shared regions as explained in section 4.2.1.2.

The remote signaling mechanism employed is in fact a variation of the channel mechanism presented in the previous section. This time though, instead of having a

pointer carrying information of the packet, a "mutant pointer" is formed and stored in the channel. This "mutant pointer" carries control information and the operation the requesting processor wishes to be performed. The process of remote signaling is described in the following steps:

1. The Universal Datagram Services layer formats a "mutant pointer" that carries control information.
2. The channel driver stores the "mutant pointer" in the appropriate channel and interrupts its neighbour that is situated between the sender and the receiver.
3. The running process at the interrupted processor is suspended and its channel driver reads the information in the channel.
4. The pointer is interpreted to be "mutant" and the interrupted processor performs the operation indicated, without passing any information to any waiting process or server process directly.

There are some restrictions, though, in employing the above remote signaling mechanism. The operations performed by the intervening processor are limited to: (1) the implementation of the spin-locks (invisible to users) in the mutual exclusion mechanism, and (2) an interrupt to the distant processor only (in which case the distant processor reads a piece of data from the middle processor and passes it to a waiting process). These restrictions are imposed because it is not safe to give the intervening processor (indirectly to the users) the power to access the memory of the distant processor.

The two mechanisms described in this section provide enough functionality for communications among processors in the array, and enable application programs to be executed on the Homogeneous Multiprocessor using the extended buses as a communication pathway [44].

4.2. Distributed Virtual Memory Management

The Virtual Memory Manager is responsible for assigning and managing virtual spaces for processes (either users or system) in collaboration with the Physical Memory Manager, as was discussed in section 3.3.6. The three types of virtual space that exist in our system are: user programs, shared regions, and global memory [24].

The first type, user programs, normally maps into local physical memory, reserving space for a Unix-like process partitioned into program (text), stack, and data areas. The implementation of such local memory management is straightforward. The other two kinds of virtual space are managed by cooperating processors in a distributed fashion. Such an arrangement enhances a user process' power to run highly distributed applications on the Homogeneous Multiprocessor The shared regions implement bounded global memory among units of three neighbouring processors that use the extended bus to communicate. The management of shared regions, including a mutually exclusive access mechanism, will be discussed in detail in section 4.2.1. The global virtual memory, on the other hand, is provided by replicating data structures distributed on a number of processors in the system, which communicate with each other either through the H-Network or the extended bus. The multiple-copy update problem inherent in replicated objects will be dealt with by the synchronisation and consistency mechanisms, and will be presented in section 4.2.2.

4.2.1. Shared Regions: Shared Memory through the Switching Network

A shared region, an abstraction of global memory shared by three adjacent processors, encapsulates a collection of data and, if desired, an implicit mechanism for mutual exclusion. It provides processes running on three adjacent processors in the array, the capability to share a common data structure for communication and synchronisation purposes. A strict definition of the shared region is a data structure during system runtime that can be *potentially* shared by three processes running on three adjacent

processors. The techniques to be presented also apply to the case involving only two sharing processes, with minor modification.

For every shared region, one could identify an owner process and up to two sharer processes (i.e., the processes running on the two immediate neighbours to which the owner process resides). The shared region resides in the owner process's memory space assigned by the local processor, and is accessible by the two sharers through the extended bus based on their processors' hardware address mapping. It is the owner process's responsibility to create and manage the shared region through system calls made available in the Virtual Memory Management layer through the User Interface.

4.2.1.1. Guarded and Unguarded Regions

There are two types of shared regions — guarded and unguarded. The guarded regions provide an implicit mechanism for mutually exclusive access to the region, while in unguarded regions this mechanism is not present. The reason for this distinction is that if the shared region is used by sharing processes for synchronisation purposes, exclusive usage of the region has to be guaranteed. The guarded regions thus provide such a service to the users transparently. On the other hand, an unguarded region can be used; however, the synchronisation part (if it is necessary) is solely the user's responsibility. The mutual exclusion mechanism utilises a set of locks that are distributed among the memories of the three neighbouring processors to achieve mutual exclusiveness, based on the concept of spin-locking, and it will be discussed in detail in section 4.2.1.2.

There are six system calls available for the management of shared regions. In the following paragraphs, the sequence from the creation to the deletion of a shared region, from the point of view of the owner, is described.

When a user process wants to create a shared region for communication with neighbouring processes, it issues the *CreateRegion* call:

reg_cap := CreateRegion (size,guarded,status)

A shared region is created with the given *size*, and a region capability *reg_cap* which points to a collection of descriptors for the shared segment is returned. (The definition of a capability in our system will be discussed in section 4.3). The *size* parameter, of course, is determined by the programmer. The Boolean *guarded* indicates whether the segment will be shared on a mutually exclusive basis. If this value is true, the locations of the locks used by the mutual exclusion protocol are also returned with *reg_cap*; otherwise, the locks are not created and the entrance to the shared region is not governed by any protocol. The shared region space is allocated by the Physical Memory Manager, while the allocation of locks is made through a pool of locks controlled by the HM-Nucleus since system start-up time. The locks are allocated both locally and at the neighbouring processors. Of course, an unguarded shared region has faster access to the shared segment, but extreme care in coding must be provided by experienced programmers. The after-effect of this system call is that a portion of the physical memory space is allotted and the physical address of the segment is made known to the caller. However, for the calling process (in fact, also for any other process) to use the shared segment, the known physical address has to be mapped into the user process's virtual space. This can be achieved by the *BindRegion* system call:

lock_cap := BindRegion (reg_cap,port,status)

The *BindRegion* call binds a region created by the *CreateRegion* call to a specific *port* in the virtual address space of the calling process. This *port* can be used by the calling process to access the information encapsulated within the region. This call returns a lock capability (*lock_cap*) which points to the *ports* of the locks. The lock capability will be used by the *EnterRegion* call to implement the mutual exclusion protocol. If the region is unguarded, the *lock_cap* is returned as null. The region capability carries a processor number on which the shared region is created. By comparing this processor number

with the processor number of the process that issues the *BindRegion*, the relative location of the region and the locks can be obtained. The *BindRegion* call also ensures that the descriptors for the shared segment and the set of locks are loaded onto the MMU. The corresponding shared segment MMU descriptor is initially disabled if the type of the *reg_cap* is guarded (the mutual exclusion algorithm will enable the descriptor upon successful access to the region), otherwise they are enabled. The loading and enabling of the MMU descriptors are carried out through functions available in the Kernel layer, specifically, *LoadDescriptor* and *EnableDescriptor*.

The separation of creating a physical space from the mapping of the virtual address facilitates the dynamic name binding procedure of the sharing processes. This involves the identification of the physical addresses of the shared segment and the locks, and also the virtual-to-physical mapping of the sharing processes. Further details of dynamic name binding involving shared regions can be found in section 4.3.3.

When a process wants to use the shared information, it issues the *EnterRegion* call which implements the mutual exclusion algorithm:

EnterRegion (lock_cap,status)

Upon successful completion, the descriptor in the MMU is enabled by using the *EnableDescriptor* call. If the region is unguarded, which is indicated in *lock_cap*, the *EnterRegion* call is transparent. By using the implicit access mechanism, the users are guaranteed the integrity of the shared region, even though the access time will be increased as a result of the extra coding. If speed is of a concern, the shared region can be created as unguarded, provided that there is no conflict in the consistency of the shared region.

Once a process has the exclusive right to the shared segment (i.e., in its critical section), no one else will be able to access (either read or write) the same segment. When the critical section is over, an *ExitRegion* call will be performed:

ExitRegion (lock_cap,status)

The shared region is released according to the mutual exclusion algorithm. However, a process using an unguarded region need not issue the *EnterRegion* and *ExitRegion* calls as they are transparent in such instances. Upon successful completion of releasing the region, the corresponding descriptor in the MMU is disabled through the *DisableDescriptor* call.

If the shared region is of no immediate interest to the active process, the *UnBindRegion* call is issued:

UnBindRegion (reg_cap,port,status)

This call marks an entry in the physical memory management table to indicate the MMU descriptor of that region is not in use any more, in case additional descriptors are needed by the Physical Memory Manager in heavy multiprogramming environment.

When the shared region is no longer needed, a user process will issue the *DestroyRegion* call to signify its intention. Once this call is issued, the process will not be able to access the *reg_cap*. A *GetName* call must be re-issued if the process has further needs in the corresponding shared region. If the issuer is a sharer, it will signal the owner of the shared region. The convention for the *DestroyRegion* call is:

DestroyRegion (reg_cap,status)

Associated with each shared region is a flag, initialised as one when the region is created. Every time a *GetName* call is issued by a neighbour (as will be discussed in section 4.3.3.1), the flag is incremented by one. A *DestroyRegion* call will decrement this flag by one. When the owner process's Physical Memory Manager finds the flag to be zero, the shared region's space will be deallocated.

4.2.1.2. Mutual Exclusion in Guarded Regions

A useful tool for synchronisation purpose is the spin-lock. If the shared region is guarded, a spin-locking mechanism will be used to limit entrance to the non-sharable (in the sense of a single reader/writer) memory. This is represented as a critical section for the sharing processes, where sensitive information is protected. A spin-lock for guarded shared memory is conventionally implemented by having the sharing processes testing and setting a single location. In our system, it means both the sharers and the owner will be accessing the owner's memory space. A straightforward implementation of the spin-lock therefore increases the interference caused by the spinning processor(s) on the owner's lock. In medium to large size critical sections, for which this mechanism is intended, the interference caused by spinning is too high a price to pay

Instead, we have opted for a similar strategy based on the concept of a modified spin-lock. In our mutual exclusion algorithm, we maintain a central semaphore through which we can ensure mutual exclusion, but a waiting processor spins within its local memory waiting for a signal from the processor which currently is in a critical section. Another advantage of using such a mechanism is that the signaling and waiting do not involve any process switching which tends to increase processing time. For large critical sections, it seems that the spinning processors are under-utilised. However, our objective here is to minimise the execution time of parallel applications. In fact, processors are an abundant resource in a multiprocessor system.

The mutual exclusion algorithm consists of two parts — key-request and key-release. These two procedures are available to the sharing processes in the Virtual Memory Management layer through the User Interface. A Pascal description of the mutual exclusion algorithm for the shared regions is shown in Figure 4.1. Our immediate applications have only a single user process running at a local node. Therefore, we have assumed that a process will remain resident in the processor and will not be swapped out until it has finished its critical section once the entrance permission has been granted.

```
program MutualExclusion;

type
        status = (on, off);
        whichprocessor = (left, right, owner);
        lockstype = array [left..owner] of status;

var
        locks : lockstype;

indivisible procedure TestAndSet (var lock : status;
                                  var oldlock : status);
{
        This procedure represents an indivisible read and write cycle
        which is implemented by the available MC68000 instruction TAS (Motorola).
}
begin
        oldlock := lock;
        lock := on;
end; {TestAndSet}

procedure KeyRequest (var locks : lockstype;
                      var me : whichprocessor);
var
        exit : Boolean;
        oldvalue : status;
begin
        if me <> owner then
                begin
                        exit := false;
                        while not exit do
                        begin
                                locks[me] := on;
                                TestAndSet(locks[owner],oldvalue);
                                if oldvalue <> off then
                                        repeat
                                                oldvalue := locks[me]
                                        until oldvalue = off
                                else exit := true;
                        end {while}
                end {then}
        else begin
                        repeat
                                TestAndSet(locks[owner],oldvalue);
                        until oldvalue = off;
                end {else}
end; {procedure KeyRequest}
```

Figure 4.1 Mutual Exclusion Algorithm for the Shared Regions

```
procedure signal (var locks : lockstype;
                  target : whichprocessor);
{
        This procedure implements the remote signaling mechanism
        as discussed in section 4.2.2
}
begin
        locks[target] := off;
end; {procedure signal}

procedure KeyRelease (var locks : lockstype;
                      var me : whichprocessor);
begin
        if me <> owner then
                begin
                        locks[owner] := off;
                        if (me = left)
                                then signal (locks,right);
                        if (me = right)
                                then signal (locks,left);
                        locks[me] := off;
                end {then}
        else
                begin
                        locks[owner] := off;
                        locks[left] := off;
                        locks[right] := off;
                end {else}
end; {procedure KeyRelease}

begin {initially}
        locks[owner] := off;
        locks[left] := off;
        locks[right] := off;
end. {program mutual exclusion}
```

Figure 4.1 Mutual Exclusion Algorithm for the Shared Regions (cont'd)

For every shared region there correspond three locks located in the local memory of the owner and the sharers respectively, as shown in Figure 4.2. These are well-known addresses allocated during the creation of a shared region and they are encapsulated in *reg_cap*. To enter a guarded region, a process has to obtain a key before it can have the exclusive access to the shared segment. Any process wishing to enter the guarded region will execute the key-request algorithm. A sharer will set its local lock, then test-and-set the central lock. If the central lock is already set, the rejected sharer will proceed to spin at its local lock. The sharer will be busy waiting until it finds its local lock reset, before initiating another attempt to get the central lock again. The owner, on the other hand, when requesting a key, simply performs the test and set operation on its local lock without interfering with any of its neighbours.

When the critical section is over, the process with the key will execute the key-release procedure. A sharer holding the key will first reset the central lock, then reset the other sharer's lock by a remote signal, as was described in section 4.1.2, and finally reset its local lock. The owner, upon releasing the key, will reset its local lock, then reset both of its neighbours' locks, even if the neighbours are not waiting.

This algorithm guarantees one and only one process (either the owner or a sharer) can access the guarded region to maintain the exclusive nature of the guarded region and it minimises interference from spinning of non-local memory, hence its complexity is justified.

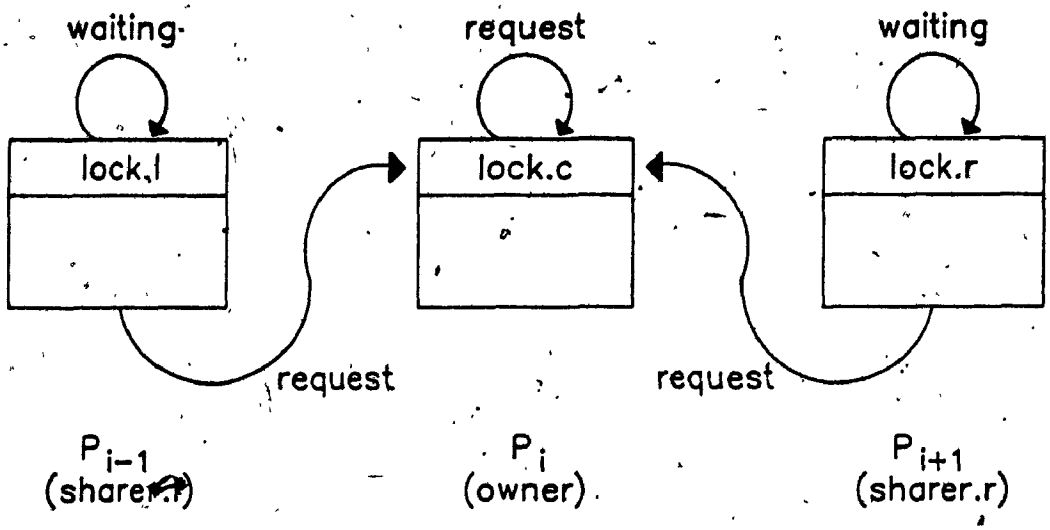


Figure 4.2 Spin-lock Implementation in Shared Regions.

4.2.2. Distributed Data Structure Management

For certain type of applications, an efficient database system is essential as it is necessary to share information among processors, i.e., to have distributed data structures. One such mechanism is implemented in the form of shared regions among groups of three neighbouring processors, as was discussed in section 4.2.1. Since there is no globally shared, physical memory in our system, we intend to support data replication, hence, globally shared data structures. In general, replicating frequently used data structures offers advantages over a single data structure server in that

i. it improves the system response time by not utilising the internode communications network, which is generally slow;

ii. it provides system reliability in case of backing up a failing node; and

iii. it increases the system capacity by avoiding the single server bottleneck particularly in cases of increased user activities.

Particular examples of the use of such global data structures include currently active file directories and the blackboard structure in the HEARSAY system [39]. These data structures may be replicated in server nodes, where they become directly accessible to the running applications.

For such a replication approach to be valid, efficient and consistent multiple-copy update mechanisms must be provided. In the subsequent sections, three such mechanisms will be presented and evaluated [43].

The following discussion regards any data structure as a data entity, be it a low level application scratch pad, or a high level file abstraction. Replicated catalogues or directories will also be treated the same as other data entities. Since migration of catalogues is not presently supported in our system, the consistency problem due to migration is not addressed in this work.

A data entity can be distributed and/or replicated on several nodes on our system. Applying principles of distributed database management, and taking into account the specific architectural features of the Homogeneous Multiprocessor, the techniques developed can be used to provide support for both user applications and the file system.

In passing, we would like to point out that it is not our purpose to treat the Homogeneous Multiprocessor as a new database architecture. Rather, by providing some minimal data structure mangement mechanisms, it serves as a base for an efficient distributed file system, and also provides a useful application abstraction to the users.

4.2.2.1. The Multiple-Copy Update Problem

A common problem in having multiple copies of a data structure at different sites is how to manage updates. Since each participating site has its own copy of the common data structure, any individual update operation may create potential deadlocks and a site may use out-of-sequence data. If updates are not serialised, participating sites may interpret the incorrect order of updates. In addition, due to the inherent delay in the communication channels, updates propagate at different rates, and the data structures in the system may not be consistent.

The solution to the multiple-copy update problem consists of two phases: synchronisation and consistency. The first phase involves the synchronisation of update requests while in the second phase, a reliable and consistent view of the global data structure is ensured. Numerous network synchronisation algorithms exist in the literature. For instances, the "bakery algorithm" proposed by Lamport generates unbounded sequence numbers to provide first-come first-served priority into critical sections [37]; Ricart and Agrawala's algorithm creates mutual exclusion in a computer network whose nodes communicate by messages [55].

A synchronisation mechanism guarantees the linear ordering of update requests to be performed one at a time only. However, due to the inherent propagation delay of the

update messages, or possible link or site failure, measures to preserve a consistent global data structure are in order. Most existing consistency-ensuring methods either update to available sites only and require acknowledgement of the update from all available sites prior to the completion of the update, or use the concept of a "primary copy" [15].

In the "primary copy" implementation, exactly one copy of an object is designated as the primary copy. All update operations are considered complete once they are applied to the primary copy. The site holding the primary copy, in turn, has the responsibility to perform the updates to all other participating sites.

In general, most methods used to solve the multiple-update problem can be categorised as one of the following:

(i) Global locking mechanisms, including the two-phase commit protocol [25]. Most of the locking mechanisms create deadlocks which must be dealt with separately through either deadlock avoidance or deadlock detection protocols; thus database management overhead tends to increase considerably.

(ii) Time stamp approaches, which are based on the ordering of events (either partial or full) [57]. Due to acknowledgement and communications protocol overhead, the efficiency of time stamp approaches decreases as the system load increases.

(iii) Circulating tokens, which are rather slow in general due to the fact that updates can be performed by one node only at any one time. However, the serialisability of updates is guaranteed [38].

(iv) Voting or majority consensus, where synchronisation is achieved based on a dialogue among participating nodes. Hence, communications overhead is rather heavy [65].

These general methods, however, have to be modified to take advantage of the particular architecture in order to be implemented efficiently. The update control mechanisms we propose are based on the observation that in certain classes of problems (such as

artificial intelligence, distributed pattern matching in expert system), a global database is read often but updated infrequently. Many applications do not require up-to-date (in terms of millisecond) information but still operate correctly with "stale" information, even though somewhat slowly (e.g., relaxation processing).

For these classes of problems, a shared data structure can be implemented as a fully-replicated database. This, in essence, simulates a virtual global memory as the Homogeneous Multiprocessor does not have any global shared memory (shared memory only exists for three adjacent processors).

A particular example using this approach is in artificial intelligence applications where a global data structure is required (e.g., the global blackboard structure in the HEARSAY speech understanding system [39]). This blackboard, as a global structure, can be fully replicated in the local memory of each node. On the other hand, if the application calls for partial replication, update mechanisms will only apply to the non-disjointed entities so that maximal parallelism can be achieved on the disjointed sets.

The methods we used for updating control map efficiently onto the architecture of the Homogeneous Multiprocessor and take advantage of the broadcasting capability of the H-Network as well as the fast interprocessor communications channel provided by the extended bus mechanism. However, the following assumptions and constraints are made to ensure the correctness of the algorithms and the suitability of applications:

(a) The targeted applications are computationally intensive, distributed applications, needing a global data structure. Typical applications in this category are Hearsay I and II, distributed simulations, expert systems, and so on.

(b) Multiple readers can operate concurrently, but only a single writer is allowed to operate at any given time. A *modify* transaction is treated as an indivisible read-and-write operation.

(c) The readers far outnumber the writers.

(d) A writer, wishing to perform an update operation, is ensured the latest version of the database. If an up-to-date read is necessary, a *modify* (X,X) can be used.

(e) Since disks are not provided to all the nodes of the Homogeneous Multiprocessor, the data structure should be small enough to fit within the available node-memory.

(f) The cache coherence problem [7] is not dealt with since there is no on-board cache at individual nodes, which our hardware does not support currently. However, this problem will have to be addressed if we upgrade to later members of the MC68000 processor family.

(g) Only a single user process runs at a node and therefore the internal consistency and integrity of the data within a single copy is preserved. The notion of consistency and integrity of the data addressed in this work is directed towards multiple copies of data distributed among participating nodes of the global data structure.

4.2.2.2. Synchronisation of Updates

The three methods proposed to achieve update synchronisation in this section, take advantage of the communication pathways available in the hardware. In all three mechanisms, a token is used to both synchronise and validate the updates.

A single token is circulated among the participating nodes of a data structure. The token carries an update sequence number, and the most recently updating node number. The arrival of the token gives a node permission to proceed with an update, immediately after a consistency check based on the information encapsulated in the token and in the previously received update messages; this will be discussed in detail in the next section. The three update synchronisation methods are differentiated by the communication pathway the token uses (either through the H-Network or the extended bus), in order to reach a node. It is assumed that the update messages are broadcast over the H-

Network.

i. Token Passing through the Extended Bus

A token is passed among a group of neighbouring nodes, each of them accessing its own copy of the same data structure. Only the node possessing the token is capable of updating, and it broadcasts the update to all the other nodes in the group through the H-Network. The token circulates among these adjacent nodes through the extended buses. A node, having received the token, uses it first for a consistency check, as explained in section 4.2.3.3. Immediately following the consistency check, the node either retains the token while it sends off its own update message (if it has one) and then passes it to its neighbour, or passes it to its neighbour immediately (if it does not possess an update message of its own).

It is assumed that the participating processes reside on a continuous array of neighbouring nodes. The token passing mechanism can be implemented by having the token passing from one end of the processor array in one direction, to the other end of the array. The boundary node at the other end then reflects the token and the process repeats itself. A variation of this bidirectional passing of the token is to have the boundary node send the token through the Network to the front of the processor array, which establishes a virtual ring structure. This alternative seems more fair as it implements a round robin scheme, while in the prior case, nodes at both ends of the array have to wait twice as much time before the token arrives again once they send it.

This method is efficient in the sense that the Network is not used for the synchronisation part, but it does have a high average startup time since the node needing to update the database must wait till the token reaches it within a bounded minimal time. The response time thus is rather slow; however, it is very efficient in a heavily-updating environment.

ii. Token-Request Broadcast through the H-Network

In this method of synchronising updates, a node wishing to perform an update will broadcast its intention on the H-Network and wait until the token is received. The node currently having possession of the token will register any requests in a first-in first-out queue of node-update-requests while all the other nodes simply ignore the request. This node-update-request queue is incorporated in the token. Again, a consistency check is performed before any updating. Update is in the form of a broadcast message through the H-Network, to all the participating nodes. The token-possessing node then deletes its own entry from the node-update-request queue, and sends the token to the next updating node registered in the queue. If the node-update-request queue is empty, the node retains the token until it receives a request from another node.

Considerable overhead is involved in this scheme. For example, the sender of the token must keep a copy of the token until an acknowledgement of its receipt by the destination node is received. This is to safeguard against loss of the token during transmission over the network. Also, a requesting station is allowed to re-request after a timeout. The possessor of the token examines the token queue and if the request has already been registered, it ignores the new request, otherwise it registers it. This is to prevent lost requests in the event that a request might be issued while the token is in transit and thus the destination node could not register the request.

To avoid potentially long delays in cases where repeated token requests coincide with the token transit period, a delayed node may also request the token during the acknowledgement cycle of the consistency check. Since the node that initiated the consistency check retains the token until all participating nodes have acknowledged, it is guaranteed that any token requests, encapsulated in the acknowledging messages, will be received and registered.

The token passing scheme using the H-Network has a fast response time and it is dynamic in nature as compared to the previous method. However, it is expensive due to the acknowledgement and re-request overhead involved.

iii. Token Controller

A third alternative to solve the multiple update problem is to assign a node as a token controller to manage update requests. The token controller maintains a queue of requesting nodes. The node at the head of the queue, once it receives the token from the controller, will proceed with the consistency check and update. Once the update is carried out, the token is returned to the token controller which passes it to the next waiting node, if any. Both updating messages and token requests utilise the network.

This method has a fast response time due to the central management by the token controller, and the overhead is handled by the token controller only. It might seem that this method of synchronisation is achieved at the expense of storage and managerial overhead at the control node; however, in a multiprocessing system, we can afford to make such a processor available, without depleting the processor resource too much. On the other hand, this structure is susceptible to system failure due to a malfunction of the token controller. A solution is to have a triplicated token controller, where three adjacent processors are assigned the task of token controlling. Together, they run collectively the same algorithms, and occasionally check on each other's integrity. A failing node of the triplet only degrades the performance of the system temporarily.

4.2.2.3. Reliable and Consistent Updates

Given the proximity of the nodes, we have assumed a *mostly* reliable communication environment. Yet, update messages are lost occasionally due to the nature of the distributed communication system. Synchronisation of multiple updates on the same data structure can be resolved by the proposed methods in the previous section. The consistency mechanisms presented in this section guarantee reliable updates to maintain system consistency. Unlike most existing approaches, no assumptions have been made that messages from one site to another will eventually be delivered or that the order of messages received is the same as the order sent.

The consistency mechanisms introduced here guarantee the updates will complete in finite time, although this time is unpredictable due to the inherent access delay to the communication network. At any point in time, all redundant copies of the data structure might be identical, or in a state of inconsistency temporarily. In the latter case, these copies are converging to the same state in finite time. Thus, the consistency of the data structure is preserved.

A link or site failure in the course of update propagation is dealt with carefully in other systems, because an acknowledgement is required from all sites before the update procedure can be completed. However, our mechanisms do not work with explicit update acknowledgements, and therefore failed sites do not affect the updating process. A failed node, upon recovery, simply requests an up-to-date version of the data structure from a working node (the node currently possessing the token). A non-failed node establishes its up-to-date version, every time it receives the token, through the process explained next.

The consistency mechanisms proceed with their operations based on the information provided by the update messages and the token. An update message, originating from the token-possessing node, encapsulates the following information: a unique sequence number, a node number, and the update message itself. The sequence number is a monotonically increasing number assigned to an update message, generated based on the information obtained from the token. The node number is the number of the node which, while possessing the token, broadcasts this update message.

The token itself consists of a unique sequence number and a node number. In addition, the token may contain a node-update-request queue as was discussed in section 4.2.2.1. The sequence number specifies the latest update message number. This sequence number is incremented and copied both to the present update message and to the token itself, by the node possessing the token (if it has an update to perform). At any given time, the sequence number found in the latest updating message broadcast

(lost or not) and the token have to be compatible, since the token is released by the updating node, after the update message has been sent by the network (assuming each node only performs a single update). Similarly, the node number found in the token corresponds to the last updating node number.

The token itself should be coded in an error detecting and correcting code to minimise the probability of corruption. In the case of a lost token, all the participating nodes will copy the most recent version of the data structure (assuming that the versions in the various nodes do not contain error apart from the missing updates). The most recent version can be determined by comparing the updating message numbers kept by the updating nodes and the sequence number of the last received messages kept by the participating nodes (as explained in the following).

Each individual node participating in the management of the distributed data structure registers the sequence number of the last received update message, and also any previous sequence numbers that are missing. The missing ones are the updating messages that have not been received and will be requested from the corresponding nodes during the consistency check.

For all three methods, each time a node receives a token, it compares the sequence number in the token with that of the last received update message. If they do not match, the node then reads the network queue and checks again. The network queue consists of incoming packets from the network and since an update message is broadcast before the token is released, then the update message should have arrived at the destination node before the token. If the two sequence numbers still do not match, then the last updating message(s) is(are) missing, and the token-possessing node requests rebroadcast from the last updating node(s).

An updating node retains its updating message until it has determined that all the participating nodes have performed the consistency check pertaining to this particular message. The condition upon which an updating node determines that its updating

message has been checked for consistency by others, is ensured by different techniques, depending on the method of synchronisation used:

i. Token Passing through the Extended-Bus

An updating node retains its update message until it receives the token again from the same neighbour, in both the bidirectional token passing and the virtual ring cases. Since all the nodes are arranged in a linear array, this condition guarantees that the token has circulated through all participating nodes, and hence each and every one of them has given the opportunity to perform a consistency check and request re-broadcast for the missing message, if applicable.

ii. Token-Request Broadcast through the H-Network

In this synchronisation method, the token reaches only nodes which have requested an update. Therefore, there is a danger that the non-requesting nodes may miss certain update messages for a prolonged period, and this would not be discovered until they received the token. To avoid such delays, a counter is kept by the token. This counter is decremented by one whenever an updating node has possession of the token. When an updating node finds that the counter has reached zero, it will broadcast a special consistency-control token to all the participating nodes. This special token forces every node to check the consistency of the updates since the previous expired count, that is, after a predetermined number of updates has been performed. This will ensure that all updates up to this point have been done. The end of the consistency check is signified when all the participating nodes have acknowledged. The originator of the consistency-control token then broadcasts an end-check message which forces the deletion of all the past update messages and announces the commencement of a new updating cycle.

iii. Token Controller

To avoid update message delays due to inactivity of certain nodes, and using the

same approach as in *ii*, the centralised token controller keeps track of the number of updates performed and goes through the same process of issuing a special consistency-control token and an end-check message..

For both *ii* and *iii*, it is understood that in between the issuing of the consistency-control tokens, nodes that receive the update token will continue to perform consistency checks. Nodes that receive out-of-sequence updates will refrain from performing their own updates until the missing update messages are received.

4.2.2.4. Multiple-Object Updates and Control Algorithms

The token mechanisms presented in the last two sections are applicable to the management of a single replicated data structure. With modification, these techniques can be expanded to accommodate a multiple-object data structure. A separate token is assigned to each data object, and updates on mutually exclusive objects can be carried out concurrently. In addition, if the objects (and their associated tokens) are linearly ordered, multiple-object updates can be achieved without deadlocks. Of course, the token then has to carry extra information about the corresponding data object, namely, the token or the object type.

All these methods have shortcomings and advantages. The method to be used in synchronising the nodes with the common data structure such that mutual exclusion is obtained before an update, is therefore dependent on the class of applications anticipated. Characteristics of applications such as transaction volume, size of data involved, reference locality, and the ratio of read transactions to the writes, have to be considered.

Multiple-protocol synchronisation schemes, such as the one found in SDD1 [57], can provide different levels of synchronisation within the same system, depending on the application and the system load. As was mentioned previously, the token passing using the extended bus is best suited for heavy rates of update requests, while the token con-

troller mechanism performs efficiently under light rates of update requests. These two mechanisms thus can be combined together to form a hybrid scheme which adapts itself to a variable demand of update requests. One way to accomplish such a scheme is to incorporate a token manager in one of the nodes in the processor array. When the token manager receives the passage of a token, it can determine the number of updates performed during the time between its possessions of the token, based on the previous and the present sequence numbers found. If within a predetermined period of time, a threshold number of updates is not reached, the rate of update requests is light. The token manager then retains the token and manages subsequent update requests. The transition from token circulation to centralised token management, as well as the location of the token manager, are made known to all the participating nodes by the token manager, in the form of a broadcast message.

4.3. Global Identifiers

Objects are continuously created, used, and destroyed by system and user programs. Each object created thus must possess a unique name by which it is known throughout its existence. From the user's point of view, the names of the objects are character strings of arbitrary length. Since such human-perceived strings are difficult to manipulate by the system, and since they cannot be guaranteed to be unique, as different users might decide to use the same string, the operating system must therefore provide support by creating unique names for the various objects it maintains and by mapping the user defined names to the unique object names created.

The foregoing discussion applies to a local node in the Homogeneous Multiprocessor. Globally, from the system point of view, matters are more complicated. Construction of distributed applications requires a uniform naming mechanism to provide an effective virtual address mapping scheme throughout the system. The unique identification provided by the naming mechanism is very useful in process scheduling,

error recovery procedures, interprocessor communications, and resource management such as file system and I/O devices. Furthermore, identifiers can be used to implement an efficient protection scheme, as the one found in Hydra where the concept of access right domain is integrated into object names [69]. Therefore, identifiers serve both the purposes of naming and protection.

In section 4.3.1, the approach we have taken for object naming is discussed, while its implementation is presented in section 4.3.2. Dynamic name binding, with two examples, are given given in section 4.3.3.

4.3.1. Capabilities

The most general form of naming in nonhierarchical protection systems is the capability. In contrast to the hierarchical or the ring protection structures which implement protection in hardware [27], capability-based protection is mostly implemented in the operating system, since most commercially available processors do not provide sufficient hardware facilities. A capability encapsulates not only a unique object name but also a type and access rights. Thus, through a capability, an object can be accessed only by authorised objects with the specified allowable operations.

In most existing operating systems, however, a capability is restricted to a unique name of the object, while some protection is provided at a lower level. For example, in the Intel 286 processor [3], segment objects can be made read only, read/write, or execute, and gates are provided for allowing only visible operations to be performed by less privileged levels. For name mapping, user-defined names and capabilities are entered in a directory or in a catalogue maintained by the operating system.

For the HM-Nucleus, since the hardware used (MC68000) does not provide privilege level protection or a full capability implementation, we have adopted the approach where a capability degenerates to a unique name [44]. Before the approach taken for the capabilities is discussed, a description of object structure in our system is in order.

Objects are arranged hierarchically in the HM-Nucleus. The highest level object is a task, which can acquire resources such as a processing node, communicate with other tasks if permitted, and execute a series of instructions for a program. A task can create processes, which are part of the running program, and they can be scheduled to run separately by the operating system. These processes in turn, can create other objects such as other tasks, processes, shared regions, data structures, buffers, and so on. A task can access objects that belong to it. It can also access the inherited resources of its child tasks. However, it cannot access directly the resources of its parent nor the resources created by its offspring. To access an object in a different family tree, a task has to get permission from the owner of that object, similar to the process of cataloguing shared objects in iRMX86 [3]. Protection, therefore can be enforced through this hierarchical structure and it can be logically assumed that the owner of an object only passes the object's capability to trustworthy parties.

A capability in the context of the HM-Nucleus consists of a string of symbols used to designate an object, including the object type, its location (the processor number), and an identification that points to the information concerning that object. Due to the distributed nature of our system, capabilities are expected to be generated in a distributed manner. In such a scheme, each local capability generator generates a name which is unique both locally and globally. Unique identifiers, or capabilities, establish object precedence and can be created without much difficulty at each local node. These local identifiers can then be mapped to the global identifiers, if desired, through the facilities available in the operating system, which enables the processors to access system resources transparently.

4.3.2. Capabilities - Implementation

At each node, capabilities can be created and mapped by using system calls available in the Table and Capabilities layers. To generate a unique name, either from a user

task or a system task, the system call *CreateName* is invoked in the Capabilities layer:

cap := CreateName (type,id)

cap is in the form of *type:processor:id*, where *type* denotes the type of object, *processor* the owner of the object, and *id* points to a collection of descriptors that contain information concerning that particular object. The uniqueness of the capability is established locally through the unique *id*, and globally due to *processor*. Under such an implementation, no same *cap* will be created due to the singularity of *id*, which can only be reused when the object is destroyed. As long as the death of an object is notified to other related processes in the system, no two objects will be referenced with the same capability.

Mapping between user names and capabilities is kept by system-maintained tables found in the Table layer. With one table per node, this can easily be expanded at a later stage to a full directory layer. To map a *named* object with the newly created capability *cap*, a *Map* call is issued by the processor which owns the object:

Map (cap,name,access)

the effect of this system call is that the correspondence between user symbolic name *name* and its capability *cap* is entered in the mapping table, according to the specified *access* right. The access right to an object can be classified in a hierarchical order as: i) universal, where all the processes within the system can access it; ii) group, where processes residing in immediate neighbouring processors have the right; iii) local, where only processes within the same node have the right; and iv) nonsharable. Similarly, an *UnMap* call by the owner deletes the *named* object from the mapping table.

UnMap (cap,name)

If the object being entered in the mapping table is a sharable object, any other tasks (either local or not) can obtain its capability by making a *GetName* call through the Table layer, provided that the access entry in the mapping table is allowed:

cap_list := GetName (name,task,processor)

where *cap_list* is the returned list of capabilities (normally, a single item list), associated with the symbolic *name* (a string of ASCII characters of arbitrary length); *task* is the capability of the task under which the capability of the *named* object was created; and *processor* is the processor on which the *named* object was created. The capability of an object is possessed by the processor on which the object is created while the task capability is passed to other cooperating tasks when this task is initiated. Currently, process migration is not allowed; however, the capability will migrate together with the object to a new processor during process migration in a future implementation. Since relocation of objects is not supported, directories or catalogues are static, and the problem of global directory consistency in the system is ignored in the current design.

4.3.3. Dynamic Name Binding

Dynamic name binding during run time supports use of multiple copies of the same object (e.g., replicated data structure and multiple generic servers), allows multiple local identifiers for the same object to achieve local autonomy, and enables many different objects to share the same name which is very useful in broadcasting. As examples of dynamic name binding, the following paragraphs will describe the name binding techniques used in shared regions and the scheme used for the mapping of global and local identifiers.

4.3.3.1. Shared Regions

We have described how a shared region is created and bound by the owner in section 4.2.1.1. However, if a sharer wants to use the newly created shared region, it must first obtain the physical location of the shared region. With the *GetName* call, a sharer of the shared region will be able to obtain the capability, hence the location of the shared region. Using the information obtained in the capability, the sharer subsequently issues the *BindRegion* call to complete the name binding process and map the region into its own virtual space.

4.3.3.2. Group/Subgroup Managers

The mapping of more than one identifiers in local levels into a unique global identifier is implemented using the concept of group/subgroup managers. All the processors in the system are divided into subgroups, and these subgroups are further divided. For example, under a group-manager, there is a subgroup of local-managers resident in each processor. Immediately above the group-managers, there is a central controller. Thus, the group-manager is in fact within a group of group-managers that are located under the same node of a tree. Such an organisation of a tree structure, embedded within the H-Network, is necessary because a single manager will become a congestion point in the Homogeneous Multiprocessor.

At each level, the Table manager holds a symbol table that maps any sublevel identifiers into identifiers of an immediate higher level. Name and address binding will be done dynamically by the use of the broadcasting facility during run time with all the managers updating their respective symbol tables.

Users at local nodes can obtain capabilities of objects located elsewhere in the system by using the *GetName* call described in section 4.3.2. Thus the following:

cap_list := GetName ('ABC',task,?)

will search for capabilities on all the processors which have been associated with the

string *'ABC'*, and the corresponding *task*, and will return a list of capabilities plus the processors where these capabilities were found. Note, however, if the access entry in the mapping table corresponding to 'ABC' indicates it is not universally available, the particular *cap* will not be returned as one of the *cap_list*.

Using the broadcast pipes, each group manager will be able to reveal its location on the tree to the others and vice versa. Local managers can then find out the location of other local managers' locations by requests traversing the tree. The location obtained in this manner is actually the processor's network address where the other local manager resides. As a result, a process at one node of the tree can converse with another located at another branch through the H-Network. However, the restriction is imposed to the effect that during the course of the conversation, processes are not allowed to migrate to another processor.

4.4. Discussion

In this chapter, the conceptual design of the operating system support for distributed applications on the Homogeneous Multiprocessor has been discussed. Extracting the distinguishing hardware features, namely, the H-Network and the extended buses, we hope to provide programmers user-friendly abstractions of such features. These supporting facilities include interprocessor communications, distributed virtual memory management, and global naming.

The interprocessor communications facility designed so far deals with communications utilising the extended bus only. More work has to be done on the pipe facility that uses the H-Network. This design decision in implementing channels and shared regions ahead of pipes is due to two reasons. First, we want to run some application programs as soon as possible to study the efficiency and feasibility of certain classes of problems on the Homogeneous Multiprocessor. These problems rely on closest neighbouring process communications, but not global information exchange. Second, the hardware design of

the H-Network is not finalised yet and the communications protocol chosen (e.g., IEEE 802) will affect our design decisions on the hardware. Currently, we are implementing certain parts of the hardware pertaining to the network functions, such as Very Large Scale Integrated (VLSI) circuit designs for network acquisition and the H-Station.

The discussion on virtual memory management in this chapter on both local and global data structures are based on the policy making modules placed in the Virtual Memory Management layer. Separated from policy, supporting mechanisms will be implemented below this layer (e.g., UDS layer), and detailed syntax and semantics of these mechanisms will be decided shortly.

So far in our HM-Nucleus implementation, we have not devoted much attention to the protection aspect. The preliminary implementation of capabilities simply consists of a unique name. Let us emphasise again that our current need is to provide enough facilities to run application programs on the processor array. The responsibility of program security thus falls on the programmer. In addition, our system is not intended for general use yet, and whoever writes application programs is expected to be familiar with the limitations of the preliminary versions of the HM-Nucleus.

5. Conclusion and Future Work

In this thesis, we have presented a performance analysis based on a simulation study for the Homogeneous Multiprocessor architecture, an operating system nucleus design with layered structure, and its support to aid distributed application programming.

Using the simulator, we studied the behaviour of the switching network under different load conditions. We concluded that the interference of a processor in using the resources of a neighbour is minimal, and there is no heavy congestion in utilising the extended buses as pathways to the shared memory modules. Also, the Homogeneous Multiprocessor was found to be a feasible architecture in an application environment. Besides being used as a tool for performance analysis, the simulator also proved to be very helpful in aiding the design and verification of application algorithms.

Implementing basic functions in the form of a nucleus provides modularity and enables application level software to be built conveniently on top of it. The layered structure of the HM-Nucleus allows information and decision encapsulation, so that software can be easily extended and modified. Separation of policy and mechanism is also made possible through this layered, modular structure.

Based on the overall nucleus design philosophy, the operating system support to be implemented will make our system available to researchers outside of our project team (user-friendly), and will bring the prominent hardware features in ways that permit their effective use by application programs. Interprocessor communications mechanisms will be provided for efficient use of the available communications pathways — the H-Network and the extended buses. Algorithms to manage shared virtual memory—the shared regions among adjacent processors and the globally replicated data structure—will give the programmer the power to use these facilities without any concern to the actual implementation details. The global naming mechanisms will let processors

communicate and synchronise effectively. With sufficient supporting functions implemented, it will enable early development of interesting applications.

We believe all the objectives set on the simulator and the operating system design are met [23]. The simulator has been used extensively by our project team in studying the Homogeneous Multiprocessor, and by other researchers in studying distributed algorithms. Implementational work has been completed and verified for the Kernel and the Physical Memory Management layers of the HM-Nucleus. Based on the functions available in these two layers, together with the overall conceptual design presented in this work, we are in the process of implementing the other layers of the nucleus and its supporting functions. In addition, there are several areas of research that can be carried out in the future:

I. Complete System Simulation

If both the H-Network simulation and the processor array simulation presented in this thesis are integrated, it will be an extremely useful tool to study the performance of the system as a truly distributed one. Such a complete system simulator possibly consists of three parts: local modules comprising of the processing unit and its memory; the switching network; and the H-Network. The event firing technique used in data flow computation can be employed for the interfacing of these three modules.

II. Benchmark

The performance evaluation of a multiprocessing system is usually based on either the speed-up achieved when the same set of problems are executed on a single node, or the linearity in effective processing power as compared to the expected one. It will be interesting to establish the performance of the Homogeneous Multiprocessor using established benchmarks comparing other systems. Such benchmarks include matrix multiplication, linear equation solving, and two-dimensional convolution [8,26].

III. Applications

The Homogeneous Multiprocessor is designed primarily to solve the classes of problems involving nearest neighbour processing, such as relaxation and image processing. However, the Homogeneous Multiprocessor might be suitable for some other less apparent applications (e.g., artificial intelligence). It will be an enriching experience to map problems onto the existing architecture. This could well be extended to the research areas of parallelism detection and problem partitioning.

IV. Protocol Verification

Informal proof of the distributed algorithms presented, such as the ones for multiple-copy consistency, can be carried out through software using protocol verification packages (e.g., Spanner). Alternatively, these algorithms can be simulated in a distributed environment; thus their efficiency can be examined. Such a simulation environment can easily be created, for example, in a network of SUN workstations with each station representing an individual node.

V. Compiler Design

Currently, all the supporting functions are available to the programmer through system calls. The utilisation of these embedded functions might be more efficient if such functions are incorporated into the programming language and generated during the compilation procedure. This will make the system more transparent and the support more readily available to the users.

These are just a few research projects which can also be performed concurrently in the present implementation stage of our project. We believe the modularity and the simplicity of the operating system nucleus and its support, and of the simulator, provide a sufficient and efficient foundation for future research development.

REFERENCES

1. *CCITT Recommendation X.25, Interface Between Data Terminal Equipment (DTE) and Data-Circuit Terminal Equipment (DCE) for Terminals Operating in the Packet Mode on Public Packet Networks*, Geneva, 1977.

2. *The MC68000 Educational Computer Board User's Manual*, Motorola Inc., Austin, Texas, 1982.

3. *iRMX86 Operating System Programmer's Reference Manual Release 6*, Intel Corporation, Santa Clara, 1984.

4. *ANSI/IEEE Standard 802.3-1985, Local Area Networks (Carrier Sense Multiple Access with Collision Detection)*, Institute of Electrical and Electronics Engineers, Inc., New York, 1985.

5. *Motorola Microprocessors Data Manual*, Motorola Inc., Austin, Texas, 1985.

6. *IEEE Trial-Use Standard 1003.1, Portable Operating System for Computer Environments*, Institute of Electrical and Electronics Engineers, Inc., New York, 1986.

7. Archibald, J. and J. L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, vol. 4, no. 4, Nov. 1986.

8. BBN Laboratories, Inc., "Benchmark Results for a 256-Node Butterfly Parallel Processor," *Computer Architecture Technical Committee Newsletter*, Sep./Dec., 1985.

9. Berry, R., et. al., *PAWS 2.0 Performance Analyst's Workbench System*, Information Research Associates, Austin, Texas, 1982.

10. Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, vol. 21, no. 11, pp. 934-941, Nov. 1978.

11. Brown, R. L., P. J. Denning, and W. F. Tichy, "Advanced Operating Systems," *IEEE Computer*, vol. 17, no. 10, pp. 173-190, Oct. 1984.

12. Butler, J. M. and A. Y. Orue, "A Facility for Simulating Multiprocessors," *IEEE Micro*, pp. 32-44, Oct. 1986.

13. Cheriton, D. R., "The V Kernel : A Software Base for Distributed System," *IEEE Software*, pp. 19-42, Apr. 1984.

14. Crowther, W., et. al., "The Butterfly Parallel Processor," *Computer Architecture Technical Committee Newsletter*, Sep./Dec. 1985.

15. Date, C. J., *Introduction to Database Systems*, II, Addition-Wesley, 1984.

16. Day, J. D. and H. Zimmermann, "The OSI Reference Model," *Proceedings of the IEEE*, vol. 71, no. 12, pp. 1334-1340, Dec. 1983.

17. Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming system," *Communications of the ACM*, vol. 11, no. 5, pp. 341-346, May 1968.

18. Dimopoulos, N. and D. Kehayas, "The H-Network: A High Speed Distributed Packet Switching Local Computer Network," *Proceedings of MELECON '83 Mediterranean Electrotechnical Conference*, p. A01.02, Athens, Greece, May 1983.

19. Dimopoulos, N. J. and K. F. Li, "Simulation and Performance Evaluation of the Homogeneous Multiprocessor Proper," *Proceedings of the 22nd Mini and Microcomputers and Their Applications*, Lugano, Switzerland, Jun. 21-23, 1983.

20. Dimopoulos, N. J., K. F. Li, and C. W. Wong, "Simulation and Performance of the Homogeneous Multiprocessor," *Proceedings of the 1984 Summer Computer Simulation Conference*, pp. 139-144, Boston, MA., Jul. 1984.

21. Dimopoulos, N. J. and C. W. Wong, "Performance Evaluation of the H-Network through Simulation," *Digital Techniques in Simulation, Communication and Control*, pp. 315-320, Elsevier Science Publishers B. V. (North-Holland), 1985.

22. Dimopoulos, N. J., "On the Structure of the Homogeneous Multiprocessor," *IEEE Transactions on Computers*, vol. C-34, no. 2, pp. 141-150, Feb. 1985.

23. Dimopoulos, N. J., K. F. Li, C. W. Wong, D. V. Ramanamurthy, and J. W. Atwood, "The Homogeneous Multiprocessor - An Overview," *The 1987 International Conference on Parallel Processing*, St. Charles, Illinois, Aug. 17-21, 1987.

24. Dimopoulos, N. J., K. F. Li, and J. W. Atwood, "Operating System Features to Support Distributed Applications in the Homogeneous Multiprocessor," *IEEE Transactions on Software Engineering*, submitted for publication.

25. Eswaran, K. P., et. al., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, vol. 19, no. 11, Nov. 1976.

26. Frenkel, K. A., "Evaluating Two Massively Parallel Machines," *Communications of the ACM*, vol. 29, no. 8, pp. 752-758, Aug. 1986.

27. Furht, B. and V. Milutinovic, "A Survey of Microprocessor Architectures for Memory Management," *IEEE Computer*, pp. 48-67, Mar. 1987.

28. Garzia, R. F., M. R. Garzia, and R. P. Zeigler, "Discrete-event Simulation," *IEEE Spectrum*, pp. 32-36, Dec. 1986.

29. Gottlieb, A. R., C. P. Kruskal, K. P. McAulitte, L. Rudolph, and M. Snir, "The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. C-32, pp. 175-190, 1983.

30. Grenblatt, E. E. and R. C. Holt, "The SUE Operating System," *Canadian Journal of Operational Research and Information Processing*, vol. 14, no. 3, pp. 227-232, Oct. 1976.

31. Hoare, C. A. R., "Monitors: an Operating System Structuring Concept," *Communications of the ACM*, vol. 17, no. 10, pp. 549-557, 1974.

32. Hoare, C. A. R., "Communication Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 667-677, Aug. 1978.

33. Holt, R. C., *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, 1983.

34. Ichbiah, J. D., et. al., "Rationale for the Design of the Ada Programming Language," *ACM SIGPLAN Notices*, vol. 14, no. 6, Jun. 1979.

35. Knuth, D. E., *The Art of Computer Programming - Fundamental Algorithms*, 1, Addison-Wesley, New York, 1968.

36. Kumar, D., "A Novel Approach to Sequential Simulation," *IEEE Software*, pp. 25-33, Sep. 1986.

37. Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453-455, Aug. 1974.

38. LeLann, G., "Algorithms for Distributed Data Sharing which Use Tickets," *Third Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 259-272, Berkeley, 1978.

39. Lesser, V. R., R. D. Fennell, L. D. Erman, and D. R. Reddy, "Organization of the Hearsay II Speech Understanding System," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 23, pp. 11-24, Feb. 1975.

40. Li, K. F., N. J. Dimopoulos, and J. W. Atwood, "An Operating System Design for the Homogeneous Multiprocessor," *Proceedings of the International Computer Symposium'86*, pp. 72-81, Tainan, Taiwan, Dec. 15-19, 1986.

41. Li, K. F., N. J. Dimopoulos, and J. W. Atwood, "The HM-Nucleus: A Distributed Operating System Nucleus for the Homogeneous Multiprocessor," *IEEE Micro*, pp. 14-24, Feb. 1987.

42. Li, K. F. and N. Dimopoulos, "The Performance Analysis of the Homogeneous Multiprocessor Proper," *Canadian Journal of Electronics*, pp. 3-10, Jan. 1987.

43. Li, K. F., N. Dimopoulos, and J. W. Atwood, "Support for Distributed Data Structures in the Homogeneous Multiprocessor," *The Second International Conference on Computer and Applications*, Beijing, China, Jun. 24-26, 1987.

44. Li, K. F., N. J. Dimopoulos, and J. W. Atwood, "Operating System Support for Distributed Applications on the Homogeneous Multiprocessor," *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA., May 3-8, 1987.

45. Marsan, M. A., G. Chiola, and G. Conte, "Generalized Stochastic Petri Net Models of Multiprocessors with Cache Memories," *Distributed Processing Technical Committee Newsletter*, vol. 7, no. 3, pp. 27-35, Nov. 1985.

46. Metcalfe, R. M. and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, vol. 19, pp. 395-404, Jul. 1976.

47. Nelson, B. J., "Remote Procedure Call," *Xerox Corporation Technical Report CSL-81-9*, May 1981.

48. Neumann, P. G., et. al., "A Provably Secure Operating System, its applications, and Proofs," Technical Report CSL-116, SRI International, May 1980.

49. Ousterhout, J. K., *Medusa: A Distributed Operating System*, UMI Research Press, Ann Arbor, 1981.

50. Panzieri, F., "Design and Development of Communication Protocols for Local Area Networks," Ph. D. Dissertation, University of Newcastle upon Tyne, 1985.

51. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.

52. Postel, J. B., *DoD Standard Transmission Control Protocol, RFC 761, IEN 129*, USC Information Sciences Institute, Jan. 1980.

53. Ramanamurthy, D. V., N. J. Dimopoulos, K. F. Li, R. V. Patel, and A. J. Al-Khalili, "Parallel Algorithms for Low Level Vision On the Homogeneous Multiprocessor," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 421-423, Miami Beach, Jun. 22-26, 1986.

54. Rashid, R. F. and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Operating Systems Review*, vol. 15, no. 5, pp. 64-754, 1981.

55. Ricart, G. and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, vol. 24, no. 1, pp. 9-16, Jan. 1981.

56. Ritchie, D. M. and K. Thompson, "The Unix Time-Sharing System," *Communications of the ACM*, vol. 17, pp. 365-375, 1974.

57. Rothnie, J. B., et. al., "Introduction to a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, vol. 5, no. 1, pp. 1-17, Mar. 1980.

58. Segal, T. L., "The TSL-2 Switching Controller," Technical Report, Concordia University, Montreal, Canada, 1986.

59. Seitz, C. L., "The Cosmic Cube," *Communications of the ACM*, vol. 28, no. 1, pp. 22-23, Jan. 1985.

60. Siegel, H. J. and J. T. Kaehn, "Parallel Image Processing/Feature Extraction Algorithms and Architecture Emulation: Interim Report for Fiscal 1981, Volume II: Architecture Emulation," Technical Report, School of Electrical Engineering, Purdue University, Oct. 1981.

61. Siegel, H. J., et. al., "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition," *IEEE Transactions on Computers*, vol. C-30, pp. 934-937, Dec. 1981.

62. Snir, M., "NETSIM-Network Simulator for the Ultracomputer," *Ultracomputer Note*, Courant Institute, NYU, New York, 1981.

63. Soloman, M. H. and R. A. Finkel, "The ROSCOE Distributed Operating System," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pp. 108-114, 1979.

64. Swan, R. J., S. J. Fuller, and D. P. Siewiorek, "Cm* - A Modular Multimicroprocessor," *Proceedings AFIPS Conference 1977*, vol. 46, pp. 645-655, 1977.

65. Thomas, R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180-209, Jun. 1979.

66. Viswanathan, R. and J. Machoul, "Quantization Properties of Transmission Parameters in Linear Predictive System," *IEEE Transactions of Acoustics Speech and Signal Processing*, vol. ASSP-23, pp. 309-321, Jun. 1975.

67. Wiley, P., "A Parallel Architecture Comes of Age at Last," *IEEE Spectrum*, vol. 24, no. 6, Jun. 1987.

68. Wong, C. W., "A Collision Free Protocol for LANs Utilizing Concurrency for Channel Contention and Transmission," M. Eng. Thesis, Concordia University, Montreal, Canada, 1985.

69. Wulf, W. A., R. Levin, and S. P. Harbison, *C.mmp - An Experimental Computer System*, McGraw-Hill, New York, 1981.

Appendix A

Indirect Call Host Assembly

The followings are the macro routines (VAX-780) for the simulator:

```
/*      SUBROUTINE ASIGN                                        */
/*                                                              */
/*      This macro routine puts the address of the              */
/*      passed argument into pointer PROCES.                    */
/*                                                              */
.data
/*                                                              */
/*      no register has to be saved.                            */
/*                                                              */
                .set            lwm1,0x000
                .globl          _a1_
_a1_:
/*                                                              */
/*      common /a1/ PROCES is used instead of var.              */
/*                                                              */
                .int            1
.text
                .globl          _asign_
_asign_:
                .word           lwm1
/*                                                              */
/*      move argument into pointer.                             */
/*                                                              */
                moval           *4(ap),*$_a1_
                ret
/*                                                              */
/*      SUBROUTINE INVOKE                                       */
/*                                                              */
/*      This macro routine invokes the subroutine which         */
/*      address is indicated by the pointer PROCES.             */
/*                                                              */
.data
                .set            lwm2,0x000
                .globl          _a1_
.text
                .globl          _invoke_
_invoke_:
                .word           lwm2
                calls           $0,*_a1_
/*                                                              */
/*      switch environment.                                     */
/*                                                              */
                ret
```

Appendix B

User interface and Monitoring Program

```
c
c  This is the main program responsible for user interface, and monitoring
c  the parallel execution of the instruction execution unit, the prefetch
c  unit, and the switching algorithm in the simulator.
c
c  all common statements are contain in the file 'block'
c
c  otfile : output file
c  nummem : memory per PE assigned (1—10000)
c  maxtim : total simulation time
c  numpro : number of processors simulated (1—64)
c  itc : diagnostic starts at this time
c  irando : 1 → program counter generated randomly by computer
c  indiv : 1 → print statistics of each PE after each run
c  isets : number of sets to be simulated
c  nops : number of nops, enter only if irando = 1.
c  iruns : number of runs in this set
c
      include 'block'
      character*30 otfile
      external wait,fetch
c
c  open input data file
c
      open(2,file='simu.dat')
      read (2,*) nummem
      read (2,*) maxtim
      read (2,*) numpro
      read (2,*) itc
      read (2,*) irando
      read (2,*) indiv
      read (2,*) isets
c
c  do isets number of sets
c
      do 100 iset=1,isets
      read (2,'(a30)') otfile
c
c  open output file
c
      open (3,file=otfile)
      write (3,10) otfile
10    format ('1',' ****  ',a30,'  ****',//)
      write (3,*) 'amount of memory assigned to PE is ',nummem
      write (3,*) 'simulation time is ',maxtim
      write (3,*) 'number of processors is ',numpro
      if (irando .eq. 0) go to 1000
c
c  determine number of NOPs in individual executable program → control
c  the interval of arrival time of requests to the switching network
c
      read (2,*) nops
      write (3,20) nops,nops+1
20    format (' numbers of nops are',i3,' and',i3)
```

```
c
c  all runs within the set will be averaged
c
1000    read (2,*) iruns
        write (3,*) ' numbers of runs in this set is',iruns
c
c  initialise system simulation clock
c
        systim=1
c
c  get WAIT address
c
        call asign(wait)
        idelok=proces
c
c  initialise boolean ic to be false indicating no diagnostics
c
        ic=0
c
c  initialises statistics variables for each set
c
        call setupstat
c
c  start simulation for one set
c
        do 200 irun = 1; iruns
c
c  initialise registers, variables, and memory
c
        call setup
        call memset
c
c  download the binary file into memory of individual PE
c
        call loadme
c
c  generate program counter randomly if indicate so,
c  else, read from input data file
c
        if (irando .eq. 0) call pcinit
        if (irando .eq. 1) call initpc
c
c  start simulation for first run
c
c  llink(ptr) : pointer to the next stage of instruction execution
c  putlnk(ptr) : pointer to the next stage of prefetch
c
        do 300 systim = 1, maxtim
c
c  check if diagnostic starts
c
        if (systim .gt. itc) ic=1
        if (ic..eq. 1) write (3,*) 'systim is',systim
c
```

```
c  instruction unit executes, for all PE in the array,
c  starting with the leftmost one
c
      do 400 ptr = 1,numpro
c
c  get current process address
c
      proces = llink (ptr)
c
c  execute current module
c
      call invoke
c
c  store next stage address
c
400   llink (ptr) = proces
c
c  prefetch unit executes, for all PE in the array,
c  starting with the leftmost one
c
      do 500 ptr = 1,numpro
      proces = putlnk(ptr)
      call invoke
500   putlnk(ptr) = proces
c
c  buffers transfer, use for synchronising the parallel activities
c
      do 600 indd=1,numpro
      jumpf(indd)=jumpb(indd)
      insmem(indd)=insmeb(indd)
      prefet(indd)=prefeb(indd)
      lidlec(indd)=lidlecb(indd)
      bgn(indd)=bgnb(indd)
      pipe1f(indd)=pip1fb(indd)
600   pipe2f(indd)=pip2fb(indd)
c
c  switching network operation
c
c  phase 1: logical closure of switches through the execution of
c  switching algorithm
c
      call switch
      do 700 ind=1,numpro
      bgok(ind)=0
c
c  phase 2: physical closure of switches in 'phase2',
c  execute only if a switch is logically closed
c
      if ((insmem(ind) .eq. 0) .and. (prefet(ind) .eq. 0)) bgok(ind)=1
      if (((bgok(ind) .eq. 1) .and. (brn(ind) .gt. 0)) .and.
     *  (bgn(ind) .eq. 1)) call phase2 (ind)
c
c  deadlock detection and recovery
c
```

```
c  case 1 and 2 of deadlocks resolved by 'arbit'
c
      logic=0
      if ((state(ind) .eq. 1) .and. (brn(ind) .gt. 0) .and.
     *  (srn(ind) .gt. 0) .and. (bgok(ind) .eq. 0) .and.
     *  (bgok(ind+1) .eq. 0) .and. (bgn(ind) .eq. 1)) logic=1
      if ((logic .eq. 1)  .and.  (memref(ind+1) .ge. nummem) .and.
     *  ((llink(ind) .eq. idelok) .or. (putlnk(ind) .eq. idelok))
     *  .and. ((llink(ind+1) .eq. idelok) .or. (putlnk(ind+1) .eq.
     *  idelok)) .and. (memref(ind+1) .lt. 2*nummem))
     *  call arbit (ind)
c
c  case 3 of deadlocks resolved by 'arbit1'
c
      logic = 0
      if ((bgn(ind) .eq. 1) .and.
     *  (brn(ind) .gt. 0) .and. (brf(ind-1) .eq. 1) .and.
     *  (srn(ind-1) .eq. 1)) logic = 1
      if ((state(ind) .eq. -1) .and. (state(ind-1) .eq. 1) .and.
     *  (memref(ind) .ge. 2*nummem) .and.
     *  (logic .eq. 1) .and. ((llink(ind) .eq. idelok) .or.
     *  (putlnk(ind) .eq. idelok)) .and. ((llink(ind-1) .eq.
     *  idelok) .or. (putlnk(ind-1) .eq. idelok)))
     *  call arbit1 (ind)
700   continue
300   continue
c
c  statistics for this run
c
      call stat
c
c  end of one run
c
200   continue
c
c  statistics for this set
c
      call aveave
      close (unit=3)
c
c  display requested memory portion
c
      call dumpmem
c
c  end of one set
c
100   continue
c
c  exit simulation
c
      close (unit=2)
      stop
      end
```

Appendix C

Sample Module of the Simulator
(Simulation of the MOVE.W instruction)

```
c
c  This is the module representing different stages of a MOVEW.W
c  instruction after it has been fetched and decoded.
c  If a reference is non-local, more time might be taken to complete
c  the cycle.
c
c  The stages in this module are implemented in two different methods.
c  The first has different submodules corresponding to individual stages
c  (i.e., movew2, movew3, and movew4) while the second one uses one
c  submodule to represent the different stages by the use of a counter
c  counting the necessary clock cycles (i.e., movew1).
c  Depending on the instruction, the first method is more elegant as
c  it shows what happens in individual stages.
c  This is used when events occur in each individual stages of a
c  particular instruction.
c  The second is more efficient as it saves programming space and is
c  suitable for stages that do not have any interaction with other
c  modules.
c
c  The instruction MOVE.W Abs.W,Dn is implemented with the first method,
c  MOVE.W Dn,(An), MOVE.W Dn,(An)+, MOVE.W (An),Dn, and MOVE.W (An)+,Dn
c  use the second one, and MOVE.W #Imm,Dn and MOVE.W An,Dn do not need
c  any extra stages.
c
c  The following MOVE.W instructions in various addressing mode are
c  implemented:
c
c  MOVE.W Abs.W,Dn
c  MOVE.W An,Dn
c  MOVE.W #Imm,Dn
c  MOVE.W Dn,(An)
c  MOVE.W Dn,(An)+
c  MOVE.W (An),Dn
c  MOVE.W (An)+,Dn
c
      SUBROUTINE movew
c
c  all common statements are in the file 'block'
c
c  oldlk(ptr) and oputlk(ptr) serve as pointers in stack to be used
c  when returning from a common submodule
c
      include 'block'
      integer decnum,byt1,byt2,byt3,byt4
c
c  movew2 - the first of the chained submodules corresponding to
c           different stages
c  movew1 - a single submodule that implements different stages
c  iidle - idle process invoked when the PE is forced to idle
c          because its local bus is being used by a neighbour
c  decode - instruction decoding process
c  addcyc - a common submodule that counts down cycles
c  chkmem - a common submodule that handles neighbouring memory
c           module access
```

```
c
      external fnovew2,iidle,decode,addcyc,chkmem,movew1
      if (ic .eq. 1) write (3,*) ptr,' is in movew1'
c
c  decode destination register #
c
      call decrn(rnd(ptr),ir(ptr,11),ir(ptr,10),ir(ptr,9))
c
c  decode source register #
c
      call decrn (rns(ptr),ir(ptr,2),ir(ptr,1),ir(ptr,0))
      if (ic .eq. 1) write (3,*) 'source reg. is',rns(ptr)
      if (ic .eq. 1) write (3,*) 'destination reg. is',rnd(ptr)
c
c  decode destination addressing mode
c
      call decmod (dmode(ptr),ir(ptr,8),ir(ptr,7),ir(ptr,6),rnd(ptr))
c
c  decode source addressing mode
c
      call decmod (smode(ptr),ir(ptr,5),ir(ptr,4),ir(ptr,3),rns(ptr))
c
c  branch if source mode is #Imm
c
      if (smode(ptr) .eq. 11) go to 1000
c
c  branch if source mode is An
c
      if (smode(ptr) .eq. 1) go to 2000
c
c  branch if source mode is Dn
c
      if (smode(ptr) .eq. 0) go to 3000
c
c  branch if source mode is (An) or (An)+
c
      if ((smode(ptr) .eq. 2).or.(smode(ptr).eq.3)) go to 4000
c
c  fall through if source mode is Abs.W
c  instruction is MOVE.W Abs.W,Dn
c
c  check if local bus is being used by neighbour (bus granted?)
c
      if (bgn(ptr) .eq. 0) go to 100
c
c  set flag to indicate the instruction execution has a
c  memory reference
c
      insmeb(ptr)=1
c
c  return here again if prefetch unit is using the local bus
c
      if ((pc(ptr) .ge. nummem) .and. (prefet(ptr) .eq. 1)) return
c
```

```
c  check if the effective address is available
c
      if (pip1fb(ptr) .eq. 1) go to 110
c
c  wait for effective address to be available'
c
      proces=llink(ptr)
      return
c
c  effective address has been fetched, get it from the pipe
c
110   call getpip
c
c  decode effective address
c
      call effadd
      if (ic .eq. 1) write (3,*) ptr,' mem. ref. is',memo(ptr)
c
c  get next stage address
c
      call asign(movew2)
c
c  if reference to neighbour memory, issue switch and bus request
c
      if ((ic .eq. 1) .and. (memo(ptr) .gt. nummem))
     *write (3,*) ' #',ptr,' issues switch request'
      call swrq (iswrqf)
c
c  check for memory violation
c
      if (iswrqf .eq. 1) go to 120
      return
c
c  local bus being used, go into idle state
c
c  put next stage address in oldlk(ptr) so that at the end of iidle,
c  the correct sequence is invoked
c
100   oldlk(ptr)=llink(ptr)
      call asign(iidle)
      return
c
c  memory reference is out of range, abort current instruction
c  reset pointer and flag
c
120   call asign(decode)
      oldlk(ptr)=proces
      insmeb(ptr)=0
      return
c
c  source mode is An
c  instruction is MOVE.W An,Dn .
c
2000  do 130 ii=1,2
```

```
c
c  move the specified An into Dn, word operation
c
130   dr(ptr,rnd(ptr),ii)=ar(ptr,rns(ptr),ii)
      if (ic .eq. 1) write (3,*) 'in movew ar,dr, dr='
      if (ic .eq. 1) write (3,*) (dr(ptr,rnd(ptr),ii),ii=1,4)
c
c  set flags according to the result of operation
c
      do 140 ii=0,3
140   sr(ptr,ii)=0
      if ((dr(ptr,rnd(ptr),1) .eq. 0) .and. (dr(ptr,rnd(ptr),2) .eq. 0))
     * sr(ptr,2)=1
      if ((dr(ptr,rnd(ptr),1) .lt. 0) .or. (dr(ptr,rnd(ptr),2) .lt. 0))
     * sr(ptr,3)=1
      if (ic .eq. 1) write (3,*) 'flgs',(sr(ptr,ii),ii=4,0,-1)
      if (ic .eq. 1) write (3,*) 'movew ar,dr finishes'
c
c  this instruction is over, decode next instruction when clock advances
c
      call asign (decode)
      oldlk(ptr) = proces
      return
c
c  source mode is #Imm
c  instruction is MOVE.W #Imm,Dn
c
c  check if immediate operand is ready
c
1000  if (pip1fb(ptr) .eq. 1) go to 150
c
c  immediate operand is not ready
c
c  check if bus is being used by neighbour
c
      if (bgn(ptr) .eq. 0) go to 100
c
c  wait for immediate operand to be available
c
      proces = llink(ptr)
      return
c
c  immediate operand is ready, get it from the pipe
c
150   call getpip
c
c  convert immediate operand to decimal form (simulator deals with
c  decimal in memory access)
c
      call immdec(decnum)
c
c  put immediate operand into appropriate bytes of a long word
c
      call parti(decnum,32,8,byt1,byt2,byt3,byt4)
```

```
c
c  move result into the specified Dn
c
      dr(ptr,rnd(ptr),1)=byt1
      dr(ptr,rnd(ptr),2)=byt2
      dr(ptr,rnd(ptr),3)=byt3
      dr(ptr,rnd(ptr),4)=byt4
c
c  set flags according to the result of operation
c
      sr(ptr,0)=0
      sr(ptr,1)=0
      sr(ptr,2)=0
      sr(ptr,3)=0
      if ((dr(ptr,rnd(ptr),1).eq.0).and.(dr(ptr,rnd(ptr),2).eq.0))
     * sr(ptr,2) = 1
      if (dr(ptr,rnd(ptr),1) .lt. 0) sr(ptr,3) = 1
      if (ic .eq. 1) write (3,*) 'flgs',(sr(ptr,ii),ii=4,0,-1)
      if (ic .eq. 1) write (3,*) 'dr#',rnd(ptr),' is '
      if (ic .eq. 1) write (3,*) (dr(ptr,rnd(ptr),ii),ii=4,1,-1)
c
c  this instruction is over, decode next instruction, when clock advances
c
      call asign (decode)
      oldlk(ptr)=proces
      insmeb(ptr)=0
      return
c
c  source mode is Dn
c  instruction is either MOVE.W Dn,(An) or MOVE.W Dn,(An)+
c
c  determine the destination effective address
c
3000  call meffadd (dmode(ptr),rnd(ptr))
      if ((ic.eq.1).and.((dmode(ptr)+1).eq.3)) write (3,10)ptr
      if ((ic.eq.1).and.((dmode(ptr)+1).eq.4)) write (3,20)ptr
10    format ('processor',i2,'is in move.w dn,(an)')
20    format ('processor',i2,'is in move.w dn,(an)+')
c
c  store next stage address in pointer
c
      call asign (movew1)
      oputlk(ptr) = proces
c
c  transfer control to common submodule that handles neighbouring
c  memory access
c
      call asign (chkmem)
      oldlk(ptr) = proces
      call chkmem
      return
c
c  source mode is (An) or (An)+
c  instruction is either MOVE.W (An),Dn or MOVE.W (An)+,Dn
```

```
c
4000 if ((ic.eq.1).and.(smode(ptr).eq.2))write(3,30)ptr
     if ((ic.eq.1).and.(smode(ptr).eq.3))write(3,40)ptr
30   format ('processor',i2,'is in move.w (an),dn')
40   format ('processor',i2,'is in move.w (an)+,dn')
c
c. xcount(ptr) to be used in extra cycle counting,
c -1 indicates negative
c
     xcount(ptr)=-1
c
c determine the source effective address
c
     call meffadd(smode(ptr),rns(ptr))
c
c store next stage address in pointer
c transfer control to common submodule that handles neighbouring
c memory access
c
     call asign (movew1)
     oputlk(ptr)=proces
     call asign (chkmem)
     oldlk(ptr)=proces
     call chkmem
     return
     end
c
c*********************************************************************
c
     SUBROUTINE movew2
     include 'block'
     external movew3
     if (ic .eq. 1) write (3,*) ptr,' is in movew2'
c
c connect to next stage
c
     call asign (movew3)
     if ((prefet(ptr) .eq. 1) .and. (brf(ptr) .eq. 0)) go to 200
     if (delay(ptr) .eq. 1) go to 210
c
c set bus request counter if requested switch is 'CLOSED'
c to compensate propagation delay of electrical signals
c
     if ((memo(ptr) .ge. nummem) .and. (memo(ptr) .lt. 2*nummem)
    * .and. (state(ptr-1) .eq. 1))
    * brc(ptr)=2
     if ((memo(ptr) .ge. 2*nummem) .and. (state(ptr) .eq. 1))
    * brc(ptr)=2
     return
c
c return here again since prefetch unit is currently using the
c local bus
c
200  proces = llink(ptr)
```

```
      delay(ptr) = 1
      return
c
c  the delay of one cycle here is to compensate local memory's availability
c
210   proces = llink(ptr)
      delay(ptr) = 0
      return
      end
c
c*********************************************************************
c
      SUBROUTINE movew3
      include 'block'
c
c  WAIT is a common submodule that waits for the neighbouring bus
c  to become available
c
      external movew4,wait
      if (ic .eq. 1) write (3,*) ptr,' in movew3'
c
c  connect to next stage
c
      call asign (movew4)
c
c  if referencing local memory, proceeds to the next cycle
c
      if (memo(ptr) .lt. nummem) return
c
c  bus request count down
c
      if (brc(ptr) .eq. 2) go to 300
c
c  set bus request counter if necessary
c
      if ((memo(ptr) .ge. nummem) .and. (memo(ptr) .lt. 2*nummem)
     * .and. (state(ptr-1) .eq. 1))
     * brc(ptr)=2
      if ((memo(ptr) .ge. 2*nummem) .and. (state(ptr) .eq. 1))
     * brc(ptr)=2
c
c  count-down and wait state for the switch to be physically closed
c
310   oldlk(ptr)=proces
      call asign (wait)
      return
300   brc(ptr)=1
      go to 310
      end
c
c*********************************************************************
c
      SUBROUTINE movew4
      include 'block'
```

```
      integer decnum,byt1,byt2,byt3,byt4
      external decode
c
c  at the end of this submodule, instruction is over and decode
c  next instruction when clock advances
c
      call asign (decode)
      if (ic .eq. 1) write (3,*) ptr, ' finish moving'
c
c  determine and adjust effective address by mapping local
c  reference into global reference
c
      call adjmem (memo(ptr),iptrs,locs)
c
c  read content of memory and convert to decimal
c
      call readme (decnum,16,iptrs,locs)
      if (ic .eq. 1) write (3,*) ptr, ' move mem ',memo(ptr),
     * ' to ',rnd(ptr)
c
c  put operand into appropriate bytes of a long word
c
      call parti (decnum,32,8,byt1,byt2,byt3,byt4)
      dr(ptr,rnd(ptr),1)=byt1
      dr(ptr,rnd(ptr),2)=byt2
      dr(ptr,rnd(ptr),3)=byt3
      dr(ptr,rnd(ptr),4)=byt4
      if (ic .eq. 1)
     * write (3,*) 'dr  ',(dr(ptr,rnd(ptr),ii),ii=4,1,-1)
c
c  set flags.
c
      do 400 ii=0,3
400   sr(ptr,ii) = 0
      if (dr(ptr,rnd(ptr),1) .eq. 0) sr(ptr,2) =1
      if (dr(ptr,rnd(ptr),1) .lt. 0) sr(ptr,3) =1
      if (ic .eq. 1)
     * write (3,*) 'flgs ',(sr(ptr,ii),ii=4,0,-1)
      insmeb(ptr)=0
c
c  if the current fetch is from neighbouring memory, reset
c  bus and switch control signals
c
      if (bgackn(ptr) .eq. 1) return
      call hsf
      return
      end
c
c*****************************************************************
c
      SUBROUTINE movew1
      include 'block'
      integer decnum,byt1,byt2,byt3,byt4
      external decode,addcyc
```

```
      if (ic .eq. 1) write (3,*) ptr, 'is in movew'
c
c determine and adjust destination memory location by mapping
c local reference into global reference
c
      call adjmem (memo(ptr),iptrd,locd)
c
c branch if source mode is Dn
c
      if (smode(ptr) .eq. 0) go to 5000
c
c determine and adjust source memory location by mapping local
c reference into global reference
c
      call adjmem (memo(ptr),iptrs,locs)
c
c read content of memory, convert to decimal
c
      call readme(decnum,16,iptrs,locs)
      if (ic .eq. 1) write (3,*) 'in movew1, decnum=',decnum
      go to 500
c
c xcount to be used for extra cycle counting,
c -1 indicates negative
c
5000  xcount(ptr) = -1
      if (ic .eq. 1) write(3,50) ptr,rns(ptr),memo(ptr)
      decnum = dr(ptr,rns(ptr),1) + dr(ptr,rns(ptr),2)*256
c
c branch if source mode is (An) or (An)+
c
500   if ((smode(ptr) .eq. 2) .or. (smode(ptr) .eq. 3)) go to 6000
c
c instruction is either MOVE.W Dn,(An) or MOVE.W Dn,(An)+
c
c store into memory
c
      call storme (decnum,16,iptrd,locd)
      if (ic .eq. 1) write (3,*) 'decnum=',decnum
c
c set flags
c
      if (decnum .eq. 0) sr(ptr,2) = 1
      if (decnum .lt. 0) sr(ptr,3) = 1
      go to 510
c
c instruction is either MOVE.W (An),Dn or MOVE.W (An)+,Dn
c
c put operand into appropriate bytes of a long word
c
6000  call parti(decnum,32,8,byt1,byt2,byt3,byt4)
      dr(ptr,rnd(ptr),1)=byt1
      dr(ptr,rnd(ptr),2)=byt2
      dr(ptr,rnd(ptr),3)=byt3
```

```
      dr(ptr,rnd(ptr),4)==byt4
c
c  instruction over, decode next instruction when clock advances
c
510   call asign (decode)
c
c  provide here in case the cycle time of these instructions increase
c
      if (xcount(ptr) .ge. 0) call asign (addcyc)
      if ((ic .eq. 1) .and. (xcount(ptr) .lt. 0)) write (3,*)
     * * ptr,' finish movew'
c
c  do post-increment or pre-decrement for corresponding addressing modes
c
      if (smode(ptr) .gt. 1) call prepost (smode(ptr),rns(ptr),16)
      if (dmode(ptr) .gt. 1) call prepost (dmode(ptr),rnd(ptr),16)
      return
 50   format(' processor ',i2,' move reg. ',i2,' to mem. ',i10)
      end
```

Appendix D

Simulated Instructions

Instructions simulated are:

add dn,dn
add an,dn
add (an),dn
and dn,dn
bcc all conditions
bra
clr (an)
clr abs.w
cmp dn,dn
cmp an,dn
cmp (an),dn
cmpa dn,an
divu dn,dn
divu an,dn
divu (an),dn
jmp abs.w
move.b dn,dn
move.b dn,(an)
move.b (an),dn
move.b (an)+,dn
move.b #imm,dn
move.b dn,(an)+
move.w dn,(an)
move.w dn,(an)+
move.w (an),dn
move.w (an)+,dn
move.w #imm,dn
move.w an,dn
move.w abs.w,dn
movea dn,an
movea #imm,an
muls dn,dn
muls an,dn
muls (an),dn
sub dn,dn
sub an,dn
sub (an),dn
tas dn
tas (an)
tas abs.w
tst (an)
tst abs.w
scc (an)
scc abs.w

Appendix E

Input and Output Specification

Input specifications are entered in files 'simu.dat', 'dump.dat', 'memorydump', 'mem.dat', and 'memory.dat', which are the default data files sought by the simulator.

'simu.dat' is used to control the simulation run. For 'simu.dat', the followings are entered in the order shown:

1. number of memory assigned to each processing element (1—10000).
2. total simulation time.
3. number of processors simulated (1—64).
4. starting time for diagnostic; if none is desired, enter as in 2.
5. a 1 will generate program counter randomly by the computer; else enter 0.
6. a 1 will print the statistics of each processing element after each run; else enter 0.
7. number of sets with the following data repeating (items 8 to 11).
8. output file name.
9. number of nops, enter only if item 5 is 1.
10. number of runs in this set (all runs in a set will be averaged).
11. starting addresses for (item 10 * item 3), i.e., runs * number of processors, enter only if item 5 is 1.

*Note: items 9 and 10 are used to control the interval arrival time of requests to the switching network.

'dump.dat' is used to specify the range of memory and the processors to be examined, after the simulation, found in the output file 'memorydump'. For 'dump.dat', enter the followings as shown:

1. total number of processors to be examined (repeat 2 accordingly).
2. processor number, starting address of memory, end address of memory (all three separated by a blank).

'mem.dat' is used to load predetermined values into the memory before simulation (otherwise, each memory location is initialised to be 0). For 'mem.dat', enter the followings as shown:

1. total number of processors to be loaded (repeat 2 accordingly).
2. processor number, starting address of memory, end address of memory (all three separated by a blank).

The desired values to be loaded into the simulated memory, according to the specification of 'mem.dat', are entered in 'memory.dat'.

The following shows a typical output, found in file specified in 8 of simu.dat:

```
**** output file name                    ****

 amount of memory assigned to PE is   10000
 simulation time is  200
 number of processors is  3
 numbers of runs in this set is  1

for each individual processor:

  Statistics for processor #  1  is
  average idle time in % is  11.3
  average wait time in % is  4.5
  average total access time in % is  21.2
  average number of access is  12
  average access time per ref. is  10.40.

overall average for all the processors in a set:

  the average of this set with  1 runs is
  min. ave. number of access is  0.666667
  max. ave. number of access is  8.666667
  ave. number of access is  3.333337
  min. ave. idle time in % is  1.66667
  max. ave. idle time in % is  18.66667
  ave. idle time in % is  12.66667
  the s.d. for idle time is  0.08333
  min. ave. wait time in % is  4.66667
  max. ave. wait time in % is  1.33337
  ave. wait time in % is  1.66667
  the s.d. for wait time is  0.02555.
  min. ave. total access time in % is  4.33333
  max. ave. total access time in % is  4.33333
  ave. total access time in % is  4.33333
  min. ave. access time per ref. is  13.0000
  max. ave. access time per ref. is  13.0000
  ave. access time per ref. is  13.0000
  the s.d. for access time per ref. is  0.
```

Appendix F

Typical Simulated Programs

To obtain the parameters in evaluating the operation of the switching network, each processor in the array executed program consisting of a DO loop during which one reference to a neighbouring memory module was made. By varying the length of the DO loop, we were able to obtain varied frequencies of requests to the switching network. Also, by varying the relative location of the requests to the neighbours within each loop, we were able to control the interarrival interval of the requests as they were presented to the switching network. The following are sample programs simulated in individual processors, for an array consisting of four processors:

```
;
; Memory space mapping for the simulator is:
; local : 0 - 9999
; left memory module : 10000 - 19999
; right memory module : 20000 - 29999
;
; processor #1
;
MEMLOCAL:   equ   1000
MEMLEFT:    equ   11000
MEMRIGHT:   equ   21000
;
LOOP:       nop
            nop
            move      #MEMLOCAL,d0
            nop
            nop
            jmp       LOOP
            end
;
; processor #2
;
MEMLOCAL:   equ   1000
MEMLEFT:    equ   11000
MEMRIGHT:   equ   21000
;
LOOP:       nop
            move      #MEMLEFT,d0
            nop
            nop
            nop
            jmp       LOOP
            end
;
; processor #3
;
MEMLOCAL:   equ   1000
MEMLEFT:    equ   11000
MEMRIGHT:   equ   21000
;
LOOP:       nop
            nop
            nop
            nop
            move      #MEMLEFT,d0
```

```
                jmp     LOOP
                end
;
; processor #4
;
MEMLOCAL:       equ     1000
MEMLEFT:        equ     11000
MEMRIGHT:       equ     21000
;
LOOP:           move    #MEMLEFT,d0
                nop
                nop
                nop
                nop
                jmp     LOOP
                end
```

Appendix G

Tunis Kernel

Tunis Kernel †

Overview of Module

Purpose

The Kernel provides the rest of the Tunis nucleus with processes, monitors and I/O primitives.

Function

The Kernel handles process switching, process scheduling and distinguishes between Tunis system processes and Tunis user processes.

The Kernel presents a passive (synchronous) I/O model to the rest of Tunis via the BeginIO, WaitIO, and EndIO primitives. The I/O model is considered to be passive because the Kernel assumes that there is always a Tunis system process waiting for every incoming interrupt.

The Kernel also contains primitives to implement Concurrent Euclid concurrency constructs such as monitors, condition queues and signals.

Heap management routines are provided to implement dynamic allocation of storage for the collection mechanism of Concurrent Euclid.

External Specifications

Imported Items

The Kernel does not reference any external modules, functions or procedures. The Kernel does, however, make use of some configuration and machine dependent constants and types such as the format of the user descriptor, and the physical limits of core memory.

Usage of Module

There are two types of entry points into the Tunis Kernel: those used to implement Concurrent Euclid primitives, and those used to implement Tunis primitives. Calls to entry points used to implement Concurrent Euclid primitives are explicitly generated by the compiler.

The Kernel entry points needed to support Concurrent Euclid are:

- *CEprocess* : called during module initialisation to create a system process descriptor for a process and to insert the system process descriptor into the ready queue.
- *CEminit* : called during monitor initialisation to set up the FIFO monitor entry queue.
- *CEcinit* : called during monitor initialisation to set up the FIFO condition queue which provides support for the signal and wait primitives.

† Extracted from the Tunis operating system documentation.

- *CEpinit* : called during monitor initialisation to set up the priority condition queue which also provides support for the signal and wait primitives.

- *CEmenter* : called to gain entrance to a monitor.

- *CEmexit* : called to exit a monitor.

- *CEsignal, CEpsignal* : called to cause the process at the head of the specified condition queue to be placed back in the ready queue. The caller is temporarily suspended outside the monitor.

- *CEwait* : called to place the caller on the FIFO condition queue specified. The caller remains there until it migrates to the front of the queue. It will then be removed on the next CEsignal to that queue.

- *CEpwait* : Called to place the caller on the priority condition queue specified. It remains there until it migrates to the front of the queue. It will then be removed on the next CEpsignal to that queue.

- *CEempty, CEpempty* : returns "true" if the specified condition queue is empty, "false" otherwise.

- *CEnew* : returns a pointer to a block of memory from the heap of the size requested. The pointer is returned with a value of nil if memory is exhausted.

- *CEfree* : releases a block of memory back into the heap. The pointer is set to nil.

A more detailed description of the construction of a Concurrent Euclid Kernel can be found in *Concurrent Euclid, The UNIX System, and Tunis* [1].

The entry points used by the Tunis nucleus are:

- *SetPriority* : used to set the run priority of the calling process.

- *GetPriority* : used to retrieve the run priority of the calling process.

- *RunUser* : used to temporarily change a system process to a user process and run the user process until it executes a trap or it is interrupted.

- *BeginIO, WaitIO, EndIO* : used to implement synchronous device interrupt handling.

- *GetDevice* : used to initialise a device interrupt handler.

- *ReadTickCount* : used by the Clock Manager to obtain a fine grained time reading.

---

[1]. Holt, R.C., *Concurrent Euclid, The UNIX System, and Tunis*, Addison-Wesley, Don Mills, Ontario., 1983, Ch. 10.

Internal Structure

The principal Kernel data structure is an array of process descriptors. A process descriptor is divided into two parts: a system process descriptor or *SPD*, and a user process descriptor or *UPD*. Both of these descriptors are highly machine dependent and are thus defined at the lowest level of Tunis - the Kernel.

The *UPD* contains information about the state of a user process, such as the contents of its registers, its program counter, stack pointer, processor status word, user process priority and run times. Storage for the *UPD*s is actually allocated in the User Manager.

The *SPD* contains information about the state of a system Concurrent Euclid process. The process' dispatch priority, the Concurrent Euclid process stack pointer, a pointer to an associated *UPD*, a pointer to other *SPD*s, the dispatch status and the system run time are the fields found in the *SPD*. The dispatch status is used to indicate whether or not a user or system process is to be dispatched.

Storage for the *SPD*s is allocated by the Kernel. The Concurrent Euclid entry points into the Kernel manipulate the *SPD*s to implement the language primitives. The Tunis entry points in the Kernel manipulate the *UPD* and allow user process to run.

To allow the Kernel to dispatch processes, all the runable processes are put in a linked list called the RunQueue. The processes with higher priorities (lower numerical numbers) are placed at the beginning of the Queue.

The Kernel is written in assembly language. The Concurrent Euclid entry points are written in assembler to gain speed although they could be coded in Concurrent Euclid. The Tunis entry points and the Kernel's interrupt handling routines are written in assembler out of necessity since the Kernel must be able to directly access the hardware registers and make use of special machine instructions.

In order to run a user process the Kernel entry point RunUser is called by the Memory Manager. This routine has one parameter; a pointer to the *UPD*. The Kernel sets up the user dispatch by switching the dispatch status in the *SPD* from a system (Envelope) process to a user process and by putting the process in the RunQueue. When this user process gets to the head of the RunQueue, the Kernel loads the machine registers and state information from the *UPD*. After calling RunUser the Envelope process is resumed when either the user process generates a trap or its time slice expires. Time slices for user processes are determined by the User Manager and when the time slice expires an internal routine called SwitchState is called. SwitchState changes the state of the running process from a user process back to a system (Envelope) process, and execution of the Envelope will be resumed.

The Kernel is also responsible for handling traps and interrupts. When a trap is caught by the Kernel a check is made to determine whether a system process or a user process caused the trap. If a user process caused the trap the user process state is saved and an internal routine called SwitchState is called. If a trap occurs while in system mode, it usually indicates a fatal error in the system code and the system halts.

Hardware Device interrupts are handled by the Kernel in order to implement Concurrent Euclid I/O primitives. Interrupts can occur during the execution of user processes, during the execution of system processes, and during the execution of most Kernel entry points. Consequently the interrupt handler is the most machine dependent and most complex part of the Kernel. After handling the interrupt it must queue the correct system process for execution in response to the interrupt, and then resume execution in whatever mode it was in before the interrupt.

The Kernel also receives interrupts from a line time clock. These interrupts occur every 1/60th of a second and are necessary to keep a fine grained clock. In the clock

interrupt handler a counter is decremented at each of these interrupts so that the Kernel can tell whenever a second has expired. The Ticker process in the Clock Manager executes a WaitIO on the line clock device. Every second this Clock process is dispatched to do its processing.

In the clock interrupt handler routine a Time Quantum is introduced and defined. In this implementation a Quantum is six clock ticks or 1/10th of a second. Another counter is used to determine whenever a Quantum expires. At each Quantum a process switch occurs regardless of whether the system is currently running a user or system process. A process switch causes the current process's state to be saved and then causes the current process to be inserted into the RunQueue according to the dispatch priority. The current process is placed after all processes in the RunQueue that have the same priority. This allows all processes running at the same priority to get some time on the CPU at regular intervals.

All Tunis system processes can control their dispatch priority through the Kernel's SetPriority and GetPriority routines.

The ReadTickCount routine is used by the Clock Manager to determine the number of ticks that has elapsed since the last second.

Appendix H

HM-Nucleus Functions Specification

********** Kernel **********

Exported calls for MMU operations:

1. **function LoadDescriptor (vir_mem : Pointer,**
status : Boolean)
return == descriptor : ShortInt

pre: BindSegment has been called to determine physical address range.

post: the descriptor indexed *descriptor* is loaded with the desired parameters in the MMU. Status is returned to indicate any error.

purpose:
load one of the 32 descriptors by transferring the content of the accumulator to the descriptor. First, the pointer *vir_mem* pointing to an area containing the variables to be loaded to the descriptor is used to index the transfer of the assigned parameters to the accumulator. Then the descriptor number is loaded into the descriptor pointer. Finally, a load descriptor operation is performed. As a result of this operation, the returned parameter *descriptor* indicates the index of the descriptor in the MMU, and a boolean variable is returned to indicate if the loading was successful.

call: WriteAccumulator, WriteSystemRegister and LoadDescriptorOp.

2. **procedure ReadDescriptor (info : Pointer,**
descriptor : ShortInt,
status : Boolean)

post: relevant information of the descriptor is loaded into an area pointed to by *info*.

purpose:
read one of the 32 descriptors by transferring the content of the descriptor into the accumulator. First, write to the descriptor pointer the number of the descriptor desired. Then perform a transfer descriptor operation. Finally, load the content of the accumulator into the area indicated by the pointer.

call: WriteSystemRegister, TransferDescriptor, and ReadAccumulator.

3. **procedure EnableDescriptor (descriptor : ShortInt,**
status : Boolean)

post: *descriptor* is enabled.

call: WriteSystemRegister.

4. procedure DisableDescriptor (descriptor : ShortInt, status : Boolean)

post: *descriptor* is disabled.

call: WriteSystemRegister.

****

Auxiliary procedures for MMU operations in the Kernel:
Note: In Concurrent Euclid notation, procedure parameters are passed through the stack while a returned parameter in a function call is stored in D0,

1. procedure ReadAccumulator

pre: accumulator has been loaded from the corresponding MMU descriptor.

post: accumulator is loaded into the area pointed to by the address passed on the stack.

purpose:
transfer the nine 8-bit registers (A0—A8) of the accumulator into the buffer pointed to by the address passed on stack (longword). This module is separated from the other read/write system registers module because a multiple move instruction is more efficient.

2. procedure WriteAccumulator

pre: a buffer pointed by the address passed on stack is loaded with required data.

post: descriptor parameters are loaded into the MMU accumulator.

purpose:
transfer the content of the buffer pointed to by the address passed on stack into the nine 8-bit registers (AC0—AC8) of the accumulator. This module is separated from the other read/write system registers module because a multiple move instruction is more efficient.

3. procedure WriteSystemRegister

pre: data (word) and address of register (longword) are pushed, in order, onto the stack.

post: the system register specified is loaded with data.

purpose:
write to system register of the MMU except the accumulator, the Interrupt Description Pointer, and the Result Descriptor Pointer. These include the address space table (AST0-AST15), the Descriptor Pointer (DP), the Interrupt Vector Register (IVR), the Global Status Register (GSR), the Local Status Register (LSR), the Segment Status Register (SSR).

4. procedure TransferDescriptor

pre: descriptor pointer (DP) has been loaded with the number of the descriptor desired.

post: the desired descriptor specified by DP is loaded into the MMU accumulator.

purpose:
In order to read the content of a descriptor, it must first be transferred into the accumulator and read from there. The transfer descriptor operation is performed by reading from the Segment Status Register.

5. function LoadDescriptorOp

pre: descriptor pointer (DP) has been loaded with the number of the descriptor desired and the accumulator is set up correctly.

post: the MMU descriptor specified by DP is loaded with parameters supplied by the accumulator, provided that there is no collision.

purpose:

load descriptor by transferring the content of the accumulator to the descriptor. A collision check is performed at the same time. A collision exists when two or more enabled descriptors are programmed to translate the same logical address. If the read operation results in a $00, the load operation is successful. If the result is a $FF, there is a collision and the load is unsuccessful. If the operation is successful, D0 is assigned a 1; else, D0 is assigned a 0.

6. function DirectTranslation

pre: the address space number (ASN) and the logical address (LA) have been loaded.

post: if successful, the physical address is loaded in the accumulator, and the descriptor pointer is loaded with the associated descriptor number.

purpose:

This function is used to directly translate the logical address into a physical address, by reading from the Direct Translation Operation Register to initiate the translation operation. It also checks if the translation is successful. If the translation is successful, information is loaded into the descriptor pointer and the accumulator. If the read operation results in a $00, the translation operation is successful, and D0 is assigned a 1. A result of $FF indicates the logical address could not be translated because it is globally undefined, and D0 is assigned a 0.

7. procedure ReadSystemRegister

pre: register address (longword) and buffer address (longword) are pushed on stack.

post: content of register is transferred to buffer.

purpose:

read system register of the MMU except the accumulator and the segment status register. These include the address space table (AST0-AST15), the Descriptor Pointer (DP), the Interrupt Vector Register (IVR), the Global Status Register (GSR), the Local Status Register (LSR), the Interrupt Description Pointer (IDP), and the Result Descriptor Pointer (RDP).

8. procedure SuperTaskSwitch

pre: supervisor data space number (word) and supervisor program space number (word) are pushed on stack in order.

post: new supervisor task is running (i.e., through AST5 and AST6 mapping of new address space).

purpose:
use a multiple move instruction in predecrement mode by writing to AST5 and AST6, the supervisor data and supervisor program address space number, followed by an illegal instruction to perform context switching between two supervisor tasks.

9. procedure UserTaskSwitch

pre: user data space number (word) and user program space number (word) are pushed on stack in order.

post: AST1 and AST2 contain new data space number and user space number, user task map through new address space.

purpose:
switch running user task by writing into AST1 and AST2 while in supervisor state.

********** Physical Memory Management **********

The Physical Memory Management Layer has the following external calls available:

1. **function AssignSegments (size : ShortInt,**
task : ShortInt,
status : Boolean)
returns = seg_num : ShortInt

pre: task and size of the segment are defined by the caller.

post: a segment descriptor is assigned with its physical memory pointer pointing to a physical memory descriptor.

purpose:
this function uses a modified Buddy algorithm (by calling auxiliary procedure Buddy) to find an available segment in the memory. If successful, it creates an entry in the segment descriptor table and sets it to active. Also, a pointer is set to the physical memory descriptor containing information of the physical space assigned: first block and last block numbers of the assigned memory, and the owner.

2. **function BindSegments (seg_num : ShortInt,**
task : ShortInt,
port : UnsignedInt,
access : ShortInt,
status : Boolean)
return = vir_mem : Pointer

pre: *AssignSegments* has been called.

post: a virtual memory descriptor is created and a pointer pointing to it is returned.

purpose:
The *BindSegments* call creates the virtual memory descriptor holding information on the virtual to physical address mapping and the access mode of the segment. *Port* is the starting virtual address of the segment passed along with *access* by the invoker. The returned parameter *vir_mem* points to this descriptor, which is used as a template for loading the MMU descriptor containing the following information: Logical Base Address (LBA), Logical Address Mask (LAM), Physical Base Address (PBA), Address Space Number (ASN), Segment Status Register (SSR), Address Space Mask (ASM), and MMU descriptor number. This procedure uses the auxiliary routines GetLAM, GetPBA, GetASN, and GetASM.

3. **procedure DeleteSegments (seg_num : ShortInt,**
status : Boolean)

post: corresponding segment descriptor, virtual memory descriptor, and physical memory descriptor are deleted, and returned to the corresponding lists. Also, the corresponding MMU descriptor is disabled.

purpose:
when a segment is no longer needed, the higher layer managers will issue this call. The corresponding entry in the segment descriptor table will be marked to indicate that the freed space can be allocated to someone else.

4. **procedure MakeSpace**

pre: an AssignSegments call returned false.

post: free list of available memory blocks is updated.

purpose:
upon unsuccessful termination of the AssignSegments call, the garbage collector examines the segments that are no longer needed (either the assigned process has been destroyed, or the packet used for communication has been consumed), and enters these available segments into a free list for future allocation. The free list is updated in such a way that at least one block of each size is available, if possible.

****

Auxiliary routines in the Physical Memory Management layer:

1. **procedure Buddy (size : ShortInt,**
first_block_no : ShortInt,
last_block_no : ShortInt,
status : Boolean)

post: returns first block and last block number of the assigned segment. If the size is not available, status is set to false.

purpose:
allocates blocks of memory using a modified Buddy algorithm. Memory is allocated starting from higher address.

2. **function GetLAM (seg_num : ShortInt)**
return = LAM : UnsignedInt

purpose:
returns the Logical Address Mask (LAM) for the MMU descriptor. LAM is obtained using the size as an index into a LAM lookup table.

3. **function GetPBA (first_block_no : ShortInt)**
return = PBA : UnsignedInt

purpose:
returns the Physical Base Address for the MMU descriptor. The least significant 8 bits are masked.

4. **function GetASN (owner : ShortInt,**
access : ShortInt)
return = ASN : ShortInt

purpose:
returns the Address Space Number (ASN) for the MMU descriptor. The current implementation has the owner number as the ASN number.

5. **function GetASM (owner : ShortInt,**
access : ShortInt)
return = ASM : ShortInt

purpose:
returns the Address Space Mask (ASM) for the MMU descriptor. The current implementation has ASM set to FF such that every bit must match for address translation.

********** Capabilities **********

1. CreateName (type : ShortInt,
id : Pointer)
return == cap : capability

pre: the pointer id is pointing to a collection of descriptors that contains information of that particular object.

post: a unique capability is created in the form of type:processor:id.

purpose:
to create a unique name by concatenating *type, processor* and *id. Type* denotes the type of object, *processor* the owner of the object, and *id* points to descriptor(s) of that object. *Type* and *id* are parameters passed by the caller.

********** Virtual Memory Management **********

Exported calls:

1. **function CreateRegion (size : ShortInt,**
guarded : Boolean,
status : Boolean)
return = reg_cap : capability

pre: *size* of the region and if it is *guarded* are determined by the caller.

post: a shared region is created with the given *size*, and a region capability *reg_cap*. If the region is *guarded*, the locations of the locks used in the mutual exclusion algorithm are also returned.

purpose:
This system call creates a shared region of the given *size* and returns a region capability *reg_cap* which points to a collection of descriptors for the shared data segment plus the locks to be used by the mutual exclusion protocol. The Boolean *guarded* indicates whether the data will be shared on a mutually exclusive basis. In the case that *guarded* = false, the locks are not created, and the data encapsulated in the region can be accessed by all three processors simultaneously. The *status* returned indicates whether the creation of the region was successful or not.

2. **function BindRegion (reg_cap : Pointer,**
port : UnsignedInt,
status : Boolean)
return = lock_cap : capability

pre: CreateRegion has been called by the owner of the shared region. *port* has been assigned by the caller.

post: lock_cap is returned.

purpose:
The *BindRegion* call binds a region created by a *CreateRegion* call to a specific *port* in the virtual address space of the calling process. This *port* can be used by the calling process to access information encapsulated within the region. This call returns a lock capability (*lock_cap*) which points to the *ports* of the locks. The lock capability will be used by the *EnterRegion* call to implement the mutual exclusion protocol. If the region is unguarded, the *lock_cap* is returned as null. The *BindRegion* call also ensures that the descriptors for the shared segment, and the set of locks are loaded onto the MMU. These descriptors are initially disabled, if the type of the *regionCap* is guarded; otherwise they are enabled. The *BindRegion* call utilises the *LoadDescriptor* and *EnableDescriptor* calls found in the Kernel to load and enable the MMU descriptors.

3. **procedure EnterRegion (Lock_cap : capability**
status : Boolean)

pre: BindRegion has been called.

post: corresponding descriptor in MMU is enabled and mutual exclusiveness is ensured if the region is guarded.

purpose:
The *EnterRegion* system call implements the mutual exclusion algorithm. Upon successful completion, it enables the MMU descriptor by using the *EnableDescriptor* call found in the Kernel MMU package. In case that the region is unguarded, which is indicated in *lock_cap*, the *EnterRegion* call is transparent.

4. **procedure ExitRegion (lock_cap : capability**
status : Boolean)

post: corresponding descriptor in MMU is disabled.

purpose:
The *ExitRegion* system call releases the shared region according to the mutual exclusion algorithm. Upon successful completion of releasing the region, it disables the MMU descriptor by using the *DisableDescriptor* call in the Kernel layer. If the region is unguarded, which is indicated in *lock_cap*, the *ExitRegion* call is transparent.

5. **procedure UnBindRegion (reg_cap : capability,**
port : UnsignedInt,
status : Boolean)

post: virtual *port* is disassociated with the region.

purpose:
The *UnBindRegion* call unbinds the *port* and the region such that a different *port* can be used by another process associating with the shared region. It also marks an entry in the physical memory management descriptor table to indicate the MMU descriptor of that region can be unloaded, in case additional address descriptors are needed by the Physical Memory Manager in a heavy multiprogramming environment.

6. **procedure DestroyRegion (reg_cap : capability,**

status : Boolean)

post: the physical space assigned to the region is deallocated.

purpose:

When the shared region is no longer needed, a user process will issue the *DestroyRegion* call to signify its intention. If the issuer is a sharer, it will signal the owner of the shared region. The last of the three *DestroyRegion* calls will invoke the owner's Physical Memory Manager and the shared region's space will be deallocated.

********** Table **********

Exported calls:

1. **function GetName (name : string,**
task : ShortInt,
processor : ShortInt)
return = cap_list : Pointer

pre: caller has the knowledge of *name*, *task*, and *processor*.

post: a list of capabilities *cap_list* is returned.

purpose:

cap_list is the returned capability list, associated with the symbolic *name*. A *cap* in the *cap_list* is in the form of type:processor:id, where type denotes the type of object, processor the owner of the object, and id points to a collection of descriptors that contain information of that particular object. *Name* is a string of ASCII characters of arbitrary length. *Task* is the capability of the task under which the capability of the *named* object was created and *processor* is the processor on which the *named* object was initially created and it currently resides. The capability of an object is possessed by the processor on which the object is created. This system call can be issued even when some of the parameters are missing or ill-defined. For example, *GetName ('ABC',task,?)* will search for capabilities on all the processors which have been associated with the string 'ABC' and will return a list of capabilities plus the processors where these capabilities were found.

2. **function Map (cap : Capability,**
name : string,
access : ShortInt)

pre: *CreateName* has been called and *access* right determined.

post: entry with corresponding *access* right in the mapping table is created.

purpose:

the effect of this system call is that the correspondence between user symbolic name *name* and its capability *cap* is entered in the mapping table, according to the specified *access* right. The access right to an object can be classified in a hierarchical order as: i) universal, where all the processes within the system can access it; ii) group, where processes residing in immediate neighbouring processors have the right; iii) local, where only processes within the same node have the right; and iv) nonsharable.

3. procedure UnMap (cap : Capability,
name : String)

pre: *Map* has been called with the same parameters *cap* and *name*.

post: the *named* object is deleted from the mapping table.

purpose:
UnMap deletes the *named* object from the mapping table.

Appendix I

MMU Address Translation

The address translation algorithm used by the MMU is:

1. The function code (FC0-FC3) is used to index into the Address Space Table to select the Cycle Address Number.
2. The Cycle Address Space Number, the logical address, and R/W are sent to each descriptor for matching. There will be a match if (i) a range match occurs, each bit position in the Logical Address Mask which is set, the incoming logical address matches the Logical Base Address; or (ii) a space match occurs, in each bit position in the Address Space Mask which is set, the Cycle Address Space Number matches the Address Space Number.
3. If there is no match, go to 7; else, continue.
4. If the segment is write-protected as indicated by the descriptor, and it is a write operation, go to 7; else, continue.
5. A physical address is generated by gating the Logical Address Mask, the Logical Address and the Physical Base Address. The logical address is passed through to the physical address in those bit positions in the Logical Address Mask which contain zeroes and the Physical Base Address is gated out in those positions which contain ones.
6. Return.
7. Generate an interrupt to the MPU indicating fault.