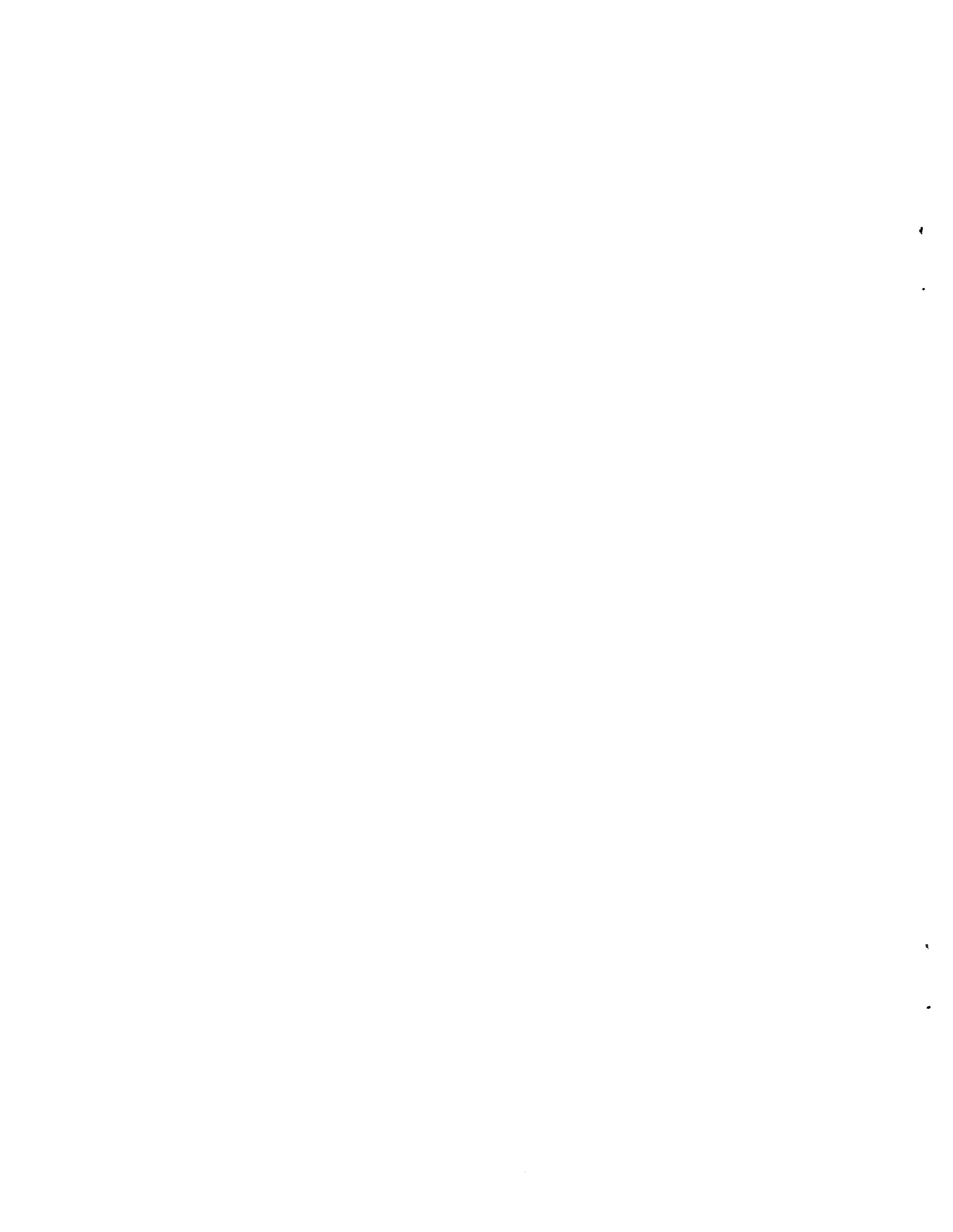
FIRMWARE STRUCTURE AND ARCHITECTURAL SUPPORT FOR MONITORS, VERTICAL MIGRATION AND USER MICROPROGRAMMING

M. Maekawa, K. Sakamura, C. Ishikawa

TR-203



/1R-a4)

Firmware Structure and Architectural Support for Monitors, Vertical Migration and User Microprogramming

Mamoru Maekawa*
Ken Sakamura**
Chiaki Ishikawa**

ABSTRACT

This paper describes firmware and hardware support necessary for constructing easy-to-understand and high performance operating systems including language translators and interpreters. Basic principles are one-to-one correspondence between logical hierarchy and physical hierarchy, and vertical migration. Implementation of monitors in firmware and architectural support for it are discussed, and a sample system is shown. Architectural support for user microprogramming is then discussed and an example is shown. After a total system firmware structure is discussed, an experiment of vertical migration is described. It is shown that a proper selection of modules for migration is extremely important. It is suggested that the direction shown in this paper is one of future directions of computer systems.

1. INTRODUCTION

Architectural support for operating systems and language translators and interpreters may vary from microprogramming to LSI chips. This paper primarily discusses microprogramming. Benefits of microprogramming are twofold; clearer hierarchical structure and higher performance. The first benefit is particularly evident for operating systems in which software is used to formulate hierarchical structures. This makes operating system structures more complex than they should be, since software of different hierarchical levels is intermixed. Operating systems are much easier to be understood and are easier to be decomposed into clear hierarchical levels if a different means of realization is used for each logical hierarchical level. Particularly, hierarchical level L_i should only be implemented based on commands available in level L_{i-1}, and should only realize commands required in level L_{i+1}. Firmware can provide one clear hierarchical level. Current machine languages do not define clear, neat, and logically natural interface to operating systems. This is causing additional complexity and difficulty in operating system construction. We should have a base language which provides a natural interface to operating systems. A base language should be determined in order to meet the requirements of operating systems.

Performance gain by microprogramming is more evident in language translators/interpreters and application programs. A basic principle is so-called "vertical migration," which shifts more frequently used functions into firmware. Higher performance is also obtained by allowing a user to tailor a computer system to his particular application by means of technique, so-called "dynamic microprogramming" or "user microprogramming."

This paper discusses issues involved in applying microprogramming techniques to operating systems and language translators/interpreters, APL interpreter in particular.

* University of Texas at Austin, Department of Computer Sciences, Austin, Texas 78712, on leave from University of Tokyo, Department of Information Science, Faculty of Science, 7-3-1 Hongo Bunkyo-ku, Tokyo, 113 Japan.

** University of Tokyo, Department of Information Science, Faculty of Science, 7-3-1 Hongo Bunkyo-ku, Tokyo, 113 Japan.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-066-4 82/03/0185 $00.75

2. OPERATING SYSTEM STRUCTURE

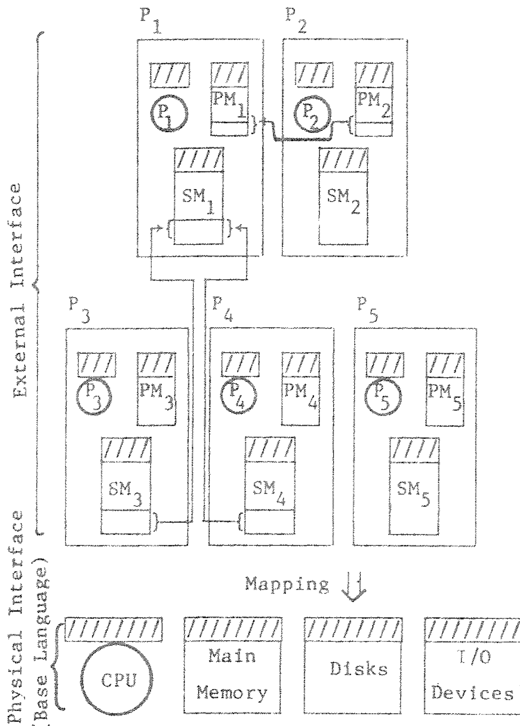
An operating system should be viewed as mapping from an external interface to a physical interface, which is called a base language in this paper. A base language should be such that it allows easy construction of an operating system and is sufficiently universal so that most operating systems can be constructed in it without increasing complexities of them more than necessary. Current machine languages are clearly far from adequate for this purpose. In order to determine a suitable base language, we must first begin with a discussion of external interface of operating systems.

2.1 External Interface

An operating system external interface consists of processes. A process itself consists of a processor, a primary logical address space, a

secondary logical address space, as shown in Figure 1. All these components define their own sets of operations allowed to them. These sets of operations are defined by the monitors of these components. Monitors also define the manners in which access is allowed.

Information can be exchanged between processes through shared portions of a logical address space and/or a secondary logical address space. In Figure 1, process P_1 allows a part of its secondary logical address space to be shared by P_3 and P_4, and a part of its primary logical address space to be shared by P_2.



P_i: Processor, PM_i: Primary Logical Address Space

SM_i: Secondary Logical Address Space

///: Monitor

Figure 1. Two Interfaces

2.2 Base Language

A base language should allow easy mapping from an external interface into it. Since sharing of physical resources and transformation of logical structures into physical structures are two major tasks of an operating system, a "monitor" is a very natural structuring unit of an operating system at the base language level. We elaborate it next.

Resources, Processes and Monitors

A dominant view on centralized operating systems is that an operating system is a collection of resources, access to which is regulated. There are three basic elements comprising an operating system in this view.

(a) Resources.
(b) Processes.
(c) Monitors.

Resources are objects to which processes make access to. Monitors govern this access. In structuring an operating system, the separation of the three basic elements; resources, processes and monitors, is most fundamental. Particularly the separation between processes and monitors is important and helpful for structuring an easy-to-understand operating system. Logically, monitors must reside in a higher hierarchical level than processes so that monitors can govern access to resources. Current computer systems do not support this hierarchy. Thus software must create this hierarchy. In current operating systems, monitors are separated from processes by making monitor routines to be non-process, non-interruptible mutually excluded routines. This can be done but causes scheduling, mutual exclusion and protection to be more complex than necessary, and consequently increases the complexity of an operating system. Separation is most easily accomplished if monitors run on a more internal level than a base language or machine instructions. Firmware provides such an ideal level. The three basic elements should thus correspond to three realization or hierarchical levels as follows:

Resources——Hardware
-------------------- ← Microinstruction interface
Monitors——Firmware
-------------------- ← Base language (machine
Processes——Software. instruction) interface

This correspondence at least solves the following three problems among others.

(a) Scheduling problem

Since a monitor is responsible for scheduling a resource to maximize its utilization, the monitor should be executed whenever the resource becomes free or a request is made. If a monitor is under the same scheduling control as processes, a special treatment of monitors is required [4]. This complicates scheduling strategies. There are essentially three approaches to cope with this problem in a conventional computer system which does not have firmware support; CPU time stealing, special processes, and special treatment of monitors [4]. The first approach treats a monitor like a machine instruction, and CPU time to execute it is stolen by a special means such as interrupts. This CPU time stealing is hidden from normal CPU dispatching. The second approach treats monitors as high priority special processes which requires only short CPU times. These processes are structured in a different way so that not only they are given higher priorities but also dispatching is easily and efficiently done. The third approach treats monitors as normal processes but assigns higher priorities to them. Among these three approaches, the first approach is most efficient but messy whereas the third approach is most systematic but least efficient. The firmware realization approach gives a clear cut between processes and non-processes. Anything written in software is process whereas non-processes are written in firmware.

(b) Mutual exclusion problem

An assurance of mutual exclusion of a monitor is tied with a transition from software to firmware, which makes control of mutual exclusion completely invisible to processes and even makes it easier.

(c) Protection of monitors

Placing monitors in firmware level automatically guarantees protection of monitors because there is no way that software can access firmware.

(d) Elimination of interrupts

Interrupts can be completely eliminated from process (or software) level. Operating system writers in a base language no longer worry about interrupts.

Two Orthogonal Modularizations

A monitor is provided for each class of resources and provides the following functions:

(a) Access operations allowed to the resources.
(b) Scheduling strategy to maximize the service and the utilization of the resources.
(c) Access validation to protect the resources from improper use.
(d) Process synchronization for interprocess communications.
(e) Deadlock detection and prevention.
(f) Time-out.

The monitor structure is schematically shown in Figure 2. We require two orthogonal modularizations. One dimension of modularization is that monitors should be independent of each other. Another dimension is that the above functions of a monitor should be independent of each other. In some cases, the same functions of different monitors need be connected. For instance, deadlock detection and prevention requires that the deadlock detection and prevention components of monitors must be connected to each other. This connection must be done without being affected by the structures of monitors concerned.

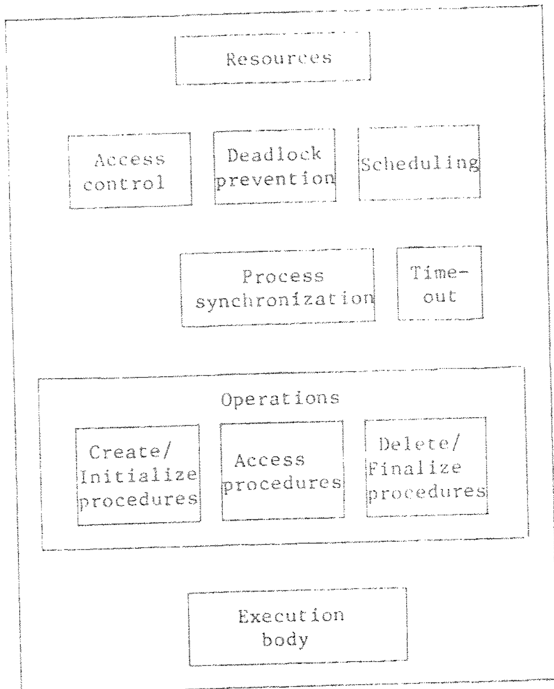


Figure 2. Monitor Structure

As software and firmware differentiate process and monitor, so firmware and LSI chips can differentiate monitor and functional component. LSI chips, which realize functions of monitor components, can provide a next natural internal level of realization.

Processing Elements for Processes and Monitors

We further extend our discussion on the scheduling problem. The scheduling problem described above arises because processes and monitors share processing elements such as CPUs. If a CPU is provided for each process and monitor, then no such scheduling problem arises, which simplifies structures of operating systems. Although this seems ridiculous, the rapid decline of hardware cost makes this realistic. In particular, providing a dedicated processor to each monitor is not an unrealistic choice.

When a monitor is responsible for active elements such as I/O devices, we have a further scheduling problem. A CPU is released after an I/O device is started and reacquired when an I/O operation is completed. In order to acquire a CPU on an I/O completion, an interrupt should be allowed to a monitor. Namely, an interrupt capability to firmware is desirable.

Interprocess Communications

Interprocess communications are made through shared resources. Therefore, any interprocess communication primitives must be a part of monitor.

3. HIGHER PERFORMANCE

Vertical Migration

A principle of performance optimization is to shift more frequently used modules to firmware and to leave less frequently used modules in software (vertical migration). This is essentially equivalent to add new machine instructions [5]. In order for this technique to be fully exploited, the following two facilities must be provided:

(a) Interrupts in firmware.
(b) Dynamic loading of firmware modules.

Interrupts in firmware are necessary because of the reasons stated in Section 2. They are also necessary because each firmware module may take a considerably longer time than a conventional machine instruction. Dynamic loading of firmware modules is necessary because the capacity of microprogram memory is limited.

Dynamic Microprogramming

A further step toward higher performance is so called "dynamic microprogramming" or "user microprogramming." This is to allow a user to write his own microprogram modules. In this way, a user can tailor a computer system to his particular needs. In order for this to be fully exploited, a computer system must provide the following additional facilities:

(a) Memory protection in microprogram memory.
(b) Two execution modes in firmware and privileged microinstructions.
(c) Program traps for exceptions and supervisor calls.

User microprogramming must be very carefully designed. One reason is that it might nullify the neatness and cleanness of operating system

structures obtained by microprogramming. Another reason is a well recognized one among manufacturers; that is, a compatibility problem. Note that vertical migration does not cause these problems as far as migration is stabilized.

In the following sections, we discuss architectural support for the requirements discussed.

4. PROCESSOR'S ARCHITECTURAL SUPPORT

In order to realize monitors easily and neatly and to obtain high performance, processors should provide the following capabilities at firmware level:

(a) Memory protection for microprogram memory.
(b) Master/slave mode.
(c) Interrupts.

At software level, on the other hand, everything is "process", which implies that there needs to be only one execution mode, there are no interrupts, and there are no privileged instructions. However, it is necessary to have a memory protection facility for main memory for two reasons:

(i) Main memory is shared among processes.
(ii) Monitors or firmware require working storage which cannot entirely be placed in microprogram memory.

Memory Protection for Microprogram Memory and Master/Slave Mode

When application users are allowed to write their own firmware, many considerations must be made at microprogramming level. For instance, application users are not always trustful. In fact, it must be taken for granted that application users make errors and even try malicious use of the system. This is quite a departure from conventional microprogramming in which only a handful of skilled designers write microprograms. Memory protection and master/slave mode mechanisms at microprogram level are necessary for detecting and stopping errors and malicious actions.

The master/slave mode is similar to that of usual multiprogramming systems in that privileged instructions are allowed only in master mode. However, its mechanism can or should be simpler at a microprogramming level than at a machine language level. For instance, EPOS [2] provides a mode mechanism which has a strong connection with the location of a microinstruction. Figure 3 shows the address space of EPOS. In this, the microprogram memory part shows the physical lay-out but the main memory part shows the logical (segmented) lay-out. Operations allowed in each mode are described below.

In master mode:

Microprogram memory (MPM): Microprograms can access (execute, read, and write except for read-only memories) any part of the microprogram memory (MPM). Note that MPM can be used as working storages.

Microinstructions: Any microinstructions including privileged microinstructions can be executed.

Main memory: Any part of the main memory (this CPU's own local memory, other CPU's local memories, shared memories and Input/Output Processor memories) can be accessed either by physical address or by logical address through the address translation mechanism (ADC).

In slave mode:

Microprogram memory (MPM): Microprograms can only execute microinstruction from only the area which lies between the addresses specified by low and high limit registers LLM and HLM (the area identified as M_S). No read and write microinstructions are allowed.

Microinstructions: Privileged microinstructions are not allowed.

Main memory: Access by logical address only is allowed.

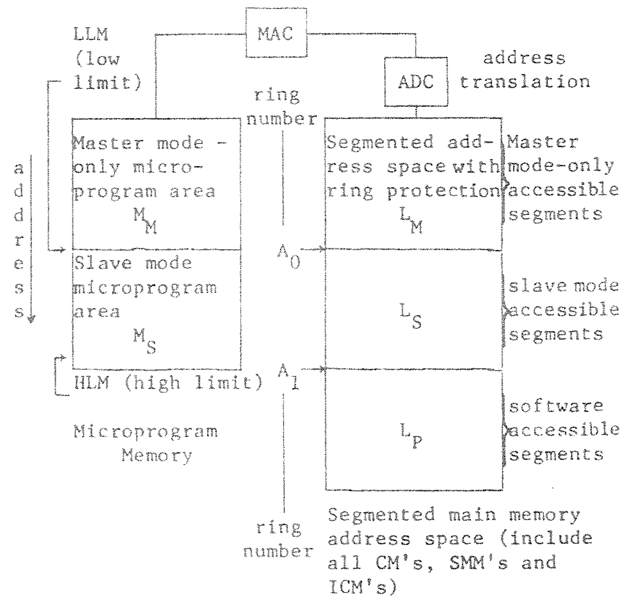


Figure 3. Address Spaces of EPOS.

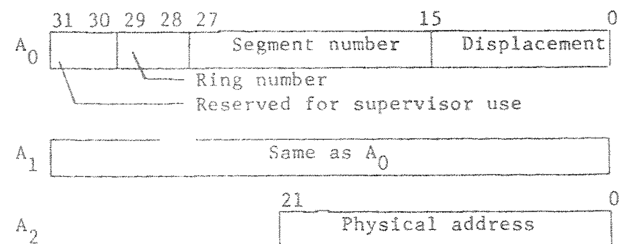


Figure 4. Address Registers.

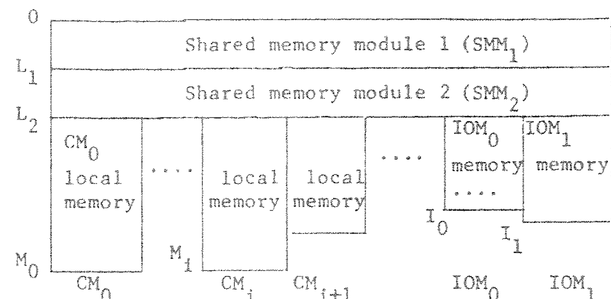


Figure 5. Physical Address Space.

Those segments having ring numbers equal to or higher than the number set up by the master mode program (the areas identified as L_S and L_P) can be executed, read and written.

A mode change is made as follows.

Master → Slave: When the microprogram instruction counter reaches equal to or beyond LLM, the mode is automatically changed to slave, and stays as so.

Slave → Master: A supervisor call (SVC) instruction changes the mode to master.

Any attempt to execute an instruction in area M_M in slave mode will cause an illegal-address interrupt. That is, programs in area M_M are always executed in master mode and those in area M_S are always executed in slave mode. LLM separates master and slave programs. The above mode-related mechanism enhances user microprogramming.

Segmented Main Memory

When an application programmer prepares firmware, he usually needs software as well, which collaborates with the firmware to perform the required task. Here, firmware means microprograms which are executed in MPM whereas software is a program in main memory. For this firmware and software collaboration, it is convenient to have a hierarchical structure in the user firmware and software. A ring protection mechanism for main memory can provide such a hierarchy as adopted in EPOS.

In EPOS, any main memory access is made through one of the three address registers A_0, A_1 and A_2. Their formats are shown in Figure 4. Register A_2 always holds a physical address and can be used only by master mode programs. Registers A_0 and A_1 can hold either a physical or logical address. Only master mode programs can use them to hold physical addresses, in which case only their low 22 bits are meaningful. As logical addresses, A_0 and A_1 are used as shown in Figure 4. Segment number and displacement fields are for segmentation and can be changed by a program of either mode. The top four bits are somewhat restricted. For A_0, only a master mode program can change them whereas the top four bits of A_1 can be changed even by a slave mode program. When an access to a segment is actually made through A_1 in slave mode, however, the larger one of the ring numbers of A_0 and A_1 is used. Thus, in slave mode, programs can never access the segments with ring numbers smaller than the one set in A_0, which is set only by a master mode program. As shown in Figure 3, the ring number of A_0 separates L_M and L_S. A hierarchy within slave mode programs can be imposed by setting some number larger than the one placed in A_0 into A_1. This boundary will usually separate software accessible segments from firmware accessible segments as shown in Figure 3. In this way, slave mode firmware can collaborate with its software (firmware serves as an interpreter) and the software can access only the limited segments. The above arrangement suggests that a microprogram must be able to access both microprogram memory and main memory. A dummy section capability is useful in a microprogram assembler.

Interrupts

As stated in Section 2, it is desirable and sometimes necessary to have interrupts at microprogramming level. They are also required for user microprogramming.

Memory protection and master/slave mode mechanisms at microprogram level for detecting and stopping errors and malicious actions require interrupts. Interrupts are also convenient for handling segment faults caused by a microprogram. They are also useful to provide an unlimited data stack in slave mode microprogramming. Flexibility in microprogram writing is increased by having interrupts. Some application users, for instance, may not want to have machine instruction fetch cycles. Without interrupts, it is difficult to use a system for a time critical application.

Unlimited Data Stack and Subroutine Stack

EPOS provides an unlimited data stack in slave mode microprogramming. One of the two 16-word register files of PULCE (Pips Universal Computing Element: a LSI chip) are designated as a stack. In slave mode, it works as an unlimited data stack using stack full and empty interrupts. The extended part of the stack is placed in main memory.

For a stack holding activation records for an ALGOL-like language, a single-segment stack is convenient because stack full and empty conditions are automatically detected by the segmentation mechanisms and because the memory address registers described before can be automatically incremented or decremented by 0, 2 or 4 at each memory reference.

For nested subroutine linkages, MCQ (microprogram sequence controller) provides a subroutine stack of 16 levels. When the stack becomes full, an interrupt occurs, but, unlike the PULCS's stack, this error is not recoverable and the program usually has to be aborted.

5. PROCESSOR AND MEMORY MANAGEMENT

A monitor performs scheduling for processors. This scheduling becomes much simpler if a physical processor can be dedicated to each process, particularly, to each monitor. Although it may not be feasible to dedicate a processor to each process, it is possible to dedicate a processor to each monitor for important resources. EPOS is such an example. EPOS is designed to be a multiprocessor system having a relatively large number (6~64) of processors to exploit this idea, in addition to many other (possibly more important) concepts and goals. The overall organization of EPOS is shown in Figure 5 while the details of each hardware module are shown in Figure 6.

System buses are multiple independent common buses, which interconnect shared memory, input/output, and computing modules. The number of system buses can grow as needed.

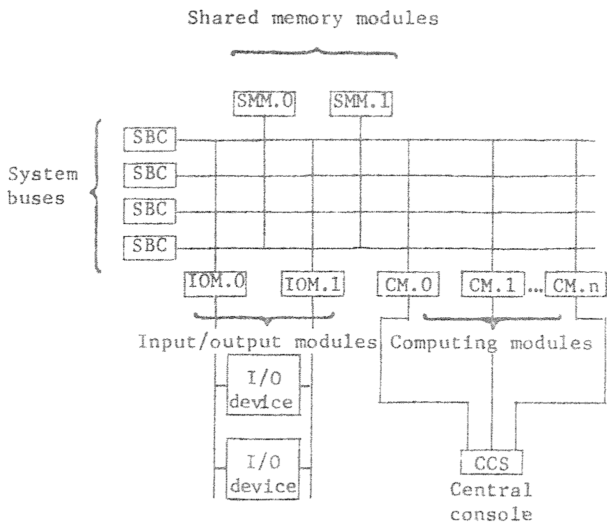


Figure 6. EPOS Overall Architecture

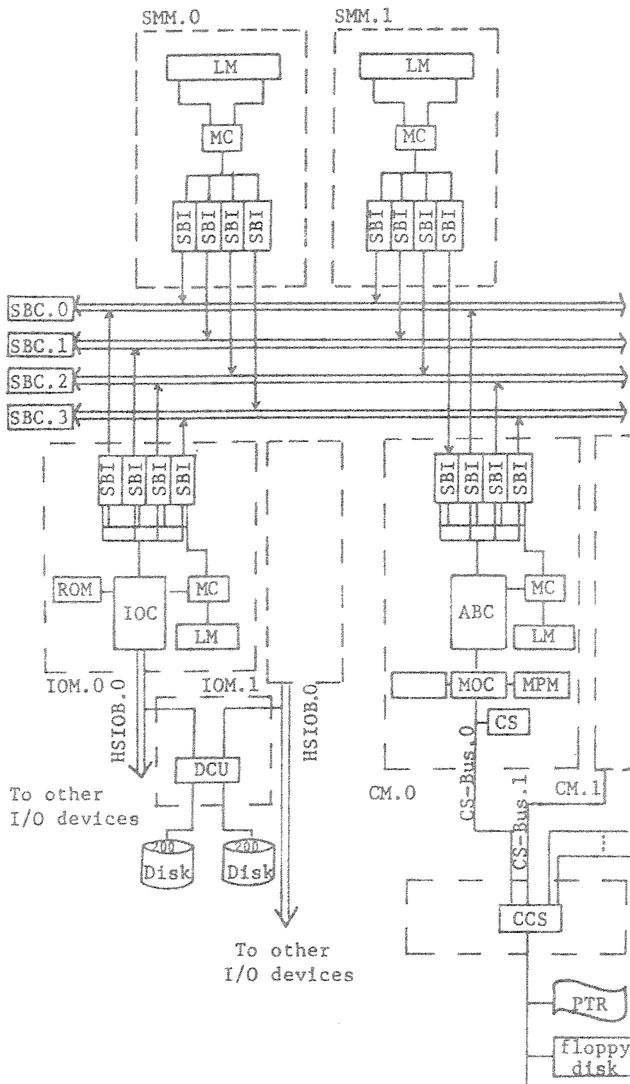


Figure 7. Structures of modules

Computing modules are responsible for computation. They can be added to the system as far as the total number of computing and input/output modules does not exceed 64. Each computing module has a computing element, microprogram memory, microprogram sequence controller, local memory and address controller for segmentation. Each computing module is expected to perform most of its computation on its own local memory, but it can also access other computing modules' local memories as well as the shared memory through the system buses.

The input/output module (IOM) is a controller for input/output buses, to which input/output devices are connected. IOM can transfer data to/from any computing module or shared memory. IOMs can also be added as far as the total number of CMs and IOMs does not exceed 64.

The shared memory modules can be used to hold data common to any module. Although the use and amount of the shared memory may depend on a particular application, the basic idea is to keep the amount of the shared memory minimal so as to avoid performance and reliability degradation.

CCS is the central console of the system for various kinds of system control such as power on/off, initial configuration set-up, dynamic configuration control, and cold and warm starts. However, CCS is designed in such a way that any faults in CCS will not cause the system to stop operation.

Assignment of processors (CMs and IOMs) in EPOS is made in such a way that scheduling problems are minimized. A processor is dedicated to each important monitoring function such as I/O supervision, file management and terminal input/output. I/O supervision is performed by IOMs while file management is performed by a CM, employing the capabilities described in Section 4. Monitoring functions for other resources such as shared data structures can also be performed by dedicated CMs. A CM is also dedicated to job scheduling, which is a more global scheduling.

Address Space and Address Translation Mechanism

The EPOS's physical address space is structured as shown in Figure 5. For computing module CM_i, low addresses $0\text{-}L_2$ specify the shared memory modules and high addresses, $L_2\text{-}M_i$, specify the CM_i's local memory. That is, each module can access other modules' local memories. This access can be made by explicitly specifying the module number of the local memory. This scheme is uniformally applied to all CMs and IOMs.

The EPOS's firmware is hierarchically structured, as shown in Figure 8. The operating system nucleus is a basic piece of firmware and is responsible for interprocess and interprocessor communications, including access to monitors.

The next level of firmware includes emulators and interpreters. An APL interpreter, for instance, accepts an APL source program, thus forming an APL machine. No ordinary machine langauge exists in the APL machine. Similarly, we can create a Pascal machine and any other commercially available machines; that is, their machine languages.

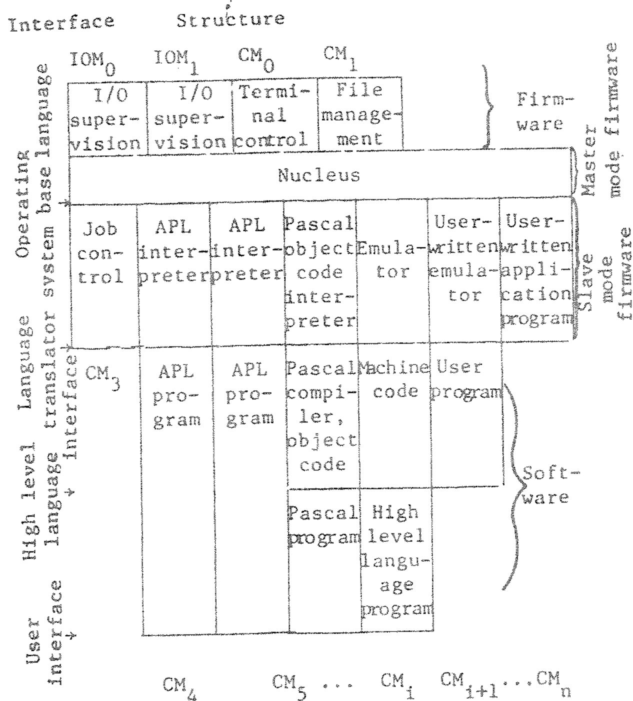


Figure 8. Operating System and Firmware Structure

Application programmers may also write their own firmware. If a user chooses to write his own emulator or interpreter, then he writes it as a slave mode microprogram and simply adds it to the firmware library. The operating system and other programs are protected from the user written firmware. A user may also choose to write his own entire application program in firmware if the program size and economy permit.

Functional Specialization and its Effects

The effects of functional specialization largely depend on the kind of application. Generally, however, about a ten-time performance improvement can be expected by direct execution by microprograms. This improvement is broken down into two-time improvement by removing machine-instruction decoding and five-time improvement by the effective execution time difference between instructions from the microprogram memory and from the main memory. Monitoring functions are microprogrammed and they obtain the above performance improvement. Numerical functions such as trigonometric and logarithmic functions can also be microprogrammed and improved. Using suitable algorithms, two-to-five time performance improvement can be expected. For more normal application requiring intermediate languages, performance improvement would be two-to-ten times.

Since functional specialization is achieved dynamically, the overhead of transferring microprograms into the microprogram memory must be taken into consideration. This transfer is made through the main memory from a secondary memory such as a magnetic disk. The effective transfer rate between the main and microprogram memories is $2*10^6$ byte/sec, which is adequate for many applications. Since no variable data are placed in the microprogram memory, no swapping-out of microprograms is necessary. Depending on the importance of microprograms, they can be stored in the main memory for fast loading.

Operating System Primitives

The time-slicing in EPOS is made by timer interrupt to a microprogram. Since almost free user microprogramming is allowed, the usual method of interrupt signal detection in each machine instruction fetch cycle is not adequate. Some emulator or interpreter may take minutes before it returns to the next fetch cycle or some interpreter may not have even the concept of fetch cycle.

One of the problems related to the design of the operating system nucleus, particularly monitors, for time-critical applications is response times for external interrupts. When monitors are implemented in machine code, they often take intolerably long times, particularly when operating systems are complex. Because of this, it is often necessary to have different operating systems for time-critical applications and for batch data processing. Multiplicity of processors and monitors in firmware help solve this problem.

7. PERFORMANCE IMPROVEMENT OF AN APL INTERPRETER BY MICROPROGRAMMING

In this section, we describe how the performance of the APL machine is improved by vertical migration. Define by r_p the ratio of firmware. That is, for some module M, r_p is defined by

$$r_p = \frac{\text{size of microprogrammed part of module M}}{\text{total size of module M}}$$

Then, r_p varies from 0 to 1.

First, as a simple case, module AFDDT, shown in Figure 9, is analyzed. Module AFDDT is one of the basic modules of the APL interpreter, and it performs a dyadic operation +, −, ×, ÷, ≤, ≠, >, ≥, <, ≤, ∧, ∨, ⍲, ⍱, |, ⌊, or ⌈. The internal structure of AFDDT is shown in Figure 10. Each node corresponds to an internal module of AFDDT. The tree shows the invocation relations among modules. An experiment shows that, when every module of AFDDT (including AFDDT itself) is microprogrammed, AFDDT executes the following number of microprogram instruction steps:

$$118n + 531$$

where n is the number of elements of the given variables. It is assumed that both operands have the same number of elements. When every module is written in APL, which is in turn interpreted by firmware, the number of microprogram execution steps is given by

$$11{,}839n + 105{,}137$$

This indicates that an APL interpretation needs about 100 times more than its equivalent microprogram.

The above cases are two extremes. For intermediate cases, Table 1 summarizes performance improvement when r_p changes. It can be seen from Table 1 that performance greatly varies depending on which modules are microprogramed. Figure 11

shows a comparison between the best and worst choices. The difference is immense. A careful selection of modules is very important. The number of repetitions also greatly affects performance. For n=10, a 10-time performance improvement requires about 60% to be microprogrammed whereas, for n=100, only 10% is necessary.

```
     ∇ V ← LARG AFDDT RARC
 [1]  1 AFFORM LARG    /*CHECK LARG*/
 [2]  2 AFFORM RARG    /*CHECK RARG*/
 [3]  AFCHK CHKTB1     /*ERROR HANDLING*/
 [4]  U1 ←, LARG       /*CHANGE TO VECTOR*/
 [5]  U2 ←, RARG       /*CHANGE TO VECTOR*/
 [6]  I ← 0            /*INITIALIZE COUNTER TO ZERO*/
 [7]  V ← MAX1 ρ 0     /*RESERVE THE RESULT AREA*/
 [8]  → (I XEQ MAX1) XCOMPS 12
                       /*REPEAT UNTIL I REACHES
                         MAX1*/
 [9]  I ← I XPLUS 1    /*INCREASE COUNTER BY 1*/
[10]  V[I] ← U1[I] ⊙ U2[I]
                       /*PERFORM SCALAR OPERATION*/
[11]  → 8              /*RETURN TO 8*/
[12]  AFDEND CHKTB1    /*CHANGE RESULT TO THE
                         APPROPRIATE TYPE*/
     ∇
```

Figure 9. Dyadic Operator Function in the Intermediate Language

AFDDT
AFFORM
AFCHK
AFDEND
AFERR
AFABS
AFERR
AFABS

Figure 10. AFDDT's Internal Structure

The above analysis is only about AFDDT. To cover a wider range of APL operations, an extensive analysis is made for two sets of operations; one for basic operations, both monadic and dyadic, +, −, ×, ÷, ⌊, ⌉, |, <, ≤, =, ≠, >, ≥, ∨, ∧, ∼, ⍲, and ⍱ and the other for, in addition to the above basic operations, array operations, ⍋, ⍒, ⍉, ⌽, ⊙/, ⊙\, for monadic operations and ⍳, ∊, /, \, ↑, ↓, ⊥, ⊤, ⊙.⊙, ∘.⊙, for dyadic operations. The assumption here is that each operation is performed once. Performance improvements are measured for scalars and arrays of 100 elements. They are shown in Figures 12 and 13. The results are similar to those of AFDDT.

As more problem oriented examples, two short programs shown in Figure 14 are analyzed. The first example is a program to sort given elements in ascending order. The second example is a program to compute sinh X. Both the programs use 9 APL system functions. The sort program executes $(19n^2 + 523n)/2$ APL lines, where n is the number of elements to be sorted. The hyperbolic-sine program requires 5 repetitions for 8 digit accuracy and executes 1,475 APL lines. Figure 15 shows the best performance improvements possible for the two programs. It also shows how one program is improved when the other is best microprogrammed. Although some discrepancy exists between the two optimals, it remains true that a small portion of program takes much of the total execution time.

r_p	Execution time in percent (100%: no microprogram)			Modules microprogrammed
	n = 1	n = 10	n = 100	
0	100.0	100.0	100.0	none
3.2	90.9	95.0	99.1	AFABS
11.1	67.5	35.5	6.19	AFDDT
19.3	87.5	93.4	98.9	AFDEND
26.6	65.0	81.6	96.8	AFFORM
30.5	55.0	28.9	5.1	AFDDT, AFDEND
37.0	(80.0)	(89.5)	(98.2)	AFCHK
37.7	32.5	17.1	3.0	AFDDT, AFFORM
45.9	52.5	75.0	95.6	AFDEND, AFFORM
48.1	47.5	25.0	4.4	AFDDT, AFCHK
57.1	20.0	10.5	1.8	AFDDT, AFFORM, AFDEND
56.3	(67.5)	(82.9)	(97.0)	AFCHK, AFDEND
63.6	45.0	71.1	95.0	AFCHK, AFFORM
67.5	35.0	18.4	3.2	AFCHK, AFDDT, AFDEND
74.7	12.5	6.6	1.2	AFDDT, AFFORM, AFCHK
82.9	(32.5)	(64.5)	(93.8)	AFCHK, AFDEND, AFFORM
100.0	0.6	0.8	1.0	all

____ (underline): best choice

( ) : worst choice

Table 1. Performance Improvement versus r_p

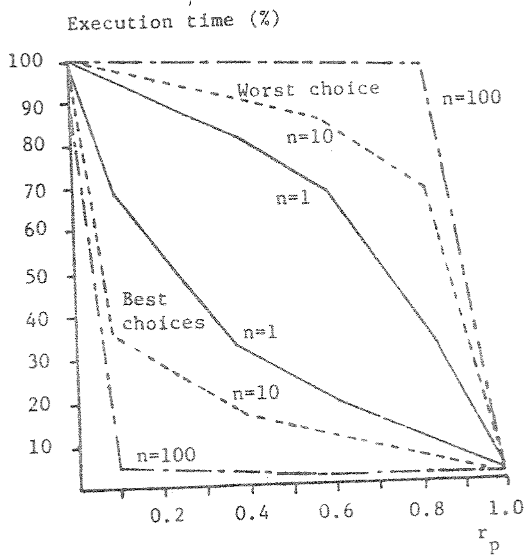
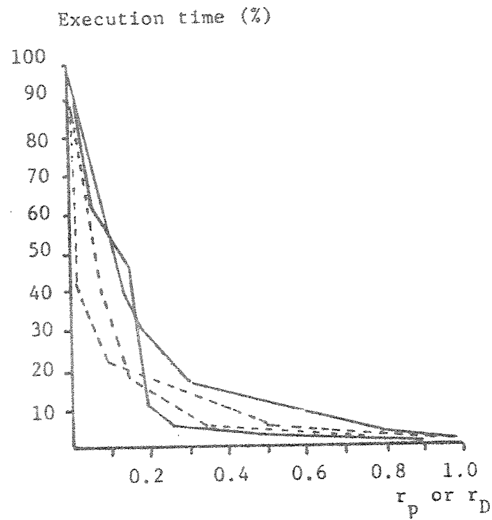
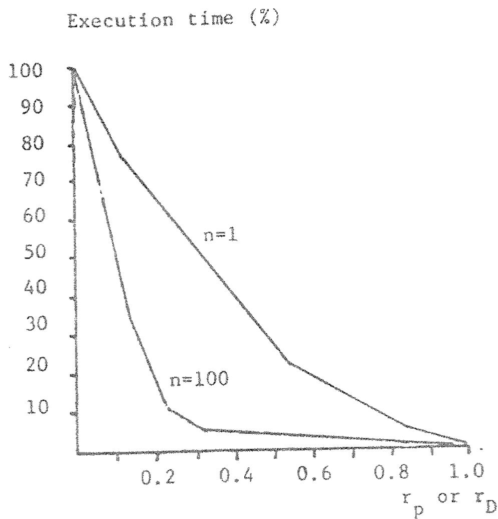


Figure 11. Performance Improvement Versus r_p



Operators included: ⍋ ⍒ ⌽ ○/ ○\ ⍳ ∊ / \ ↑ ↓
⊥ ⊤ ∘.[illegible] ∘.[illegible]
in addition to those included in Figure 6.

Figure 13. Performance Improvements for APL Operations.



Operators included:

Monadic + - × ÷ ⌊ ⌈ | ~

Dyadic + - × ÷ ⌊ ⌈ | = ≠ < ≤ > ≥ ∨ ∧ ⍱ ⍲

Figure 12. Performance Improvements for Basic Operations

```
    ∇ R    SORT; A; B; C; D
[1]   B ← ρ, A ⎕
[2]   R ← ⍳0
[3]   D ← ρ, A
[4]   A ← (A = C ← ⌊/A)/A
[5]   R ← R, (D - ρ, A) ρ C
[6]   → (B ≠ ρR)/3
[7] ∇
```

(a) Sort

```
      R ← SINH X; N; Y
[1]   N ← 1 + R ← 0
[2]   → (R = Y ← R + (X * N) ÷ !N)/0
[3]   R ← Y
[4]   N ← N + 2
[5]   → 2
[6] ∇
```

(b) Sinh X

Figure 14. Sample Programs

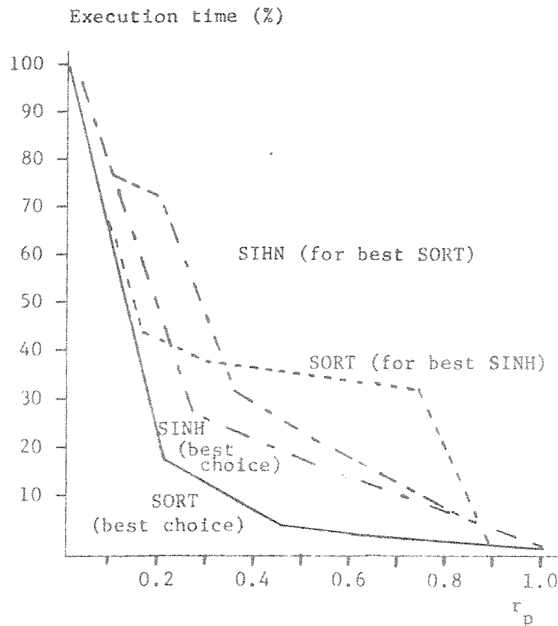


Figure 15. Sort and Sinh Performance Improvements

5. CONCLUSIONS

This paper has described firmware and hardware support for constructing easy-to-understand and high performance operating systems including language translators and interpreters. Increasing software cost coupled with decreasing hardware cost necessitates that system development burdens are to be shifted toward hardware and firmware. Particularly, the simplification of operating system structures through one-to-one correspondence between logical and physical hierarchical levels is important. Firmware plays a great role for this purpose. Another great role of firmware is functional specialization or personal tuning of processors to a wide variety of application as exemplified by dynamic microprogramming or user microprogramming.

Systems, such as EPOS, aim at utilizing rapidly progressing LSI technologies. A basic principle is that each processor is dedicated to a particular task so that designers are relieved of CPU scheduling problems. Another principle is that monitors are constructed based on internal components which are made of LSI chips so that each monitor is simply constructed by combining them. Although most current computers do not have enough processors or convenient LSI chips to completely accomplish these goals, the progress of LSI technologies will make this possible. The one-to-one correspondence of a process and a processor is helpful to simplify operating system structures. We foresee that a computer system is comprised of LSI modules which support monitors as well as hardware components.

REFERENCES

[1] M. Maekawa, I. Yamazaki, et al, Experimental Polyprocessor System--Architecture, Proc. International Symposium on Computer Architecture, Philadelphia, PA, 1979.

[2] M. Maekawa, I. Yamazaki, et al, Experimental Polyprocessor System--Operating System, Proc. International Symposium on Computer Architecture, Philadelphia, PA, 1979.

[3] M. Maekawa and Y. Morimoto, Performance Adjustment of an APL Interpreter, Microprocessors and their Applications, J. Tiberghien, G. Carlstedt, J. Lewi (eds.), North-Holland Publishing Company, 1979.

[4] M. Maekawa, Evaluation and Comparison of Process Coordination Mechanisms, Technical Report 80-23, Department of Information Science, Faculty of Science, University of Tokyo, Hongo, Tokyo, Japan, 1980.

[5] K. Sakamura, et al, Automatic Tuning of Computer Architecture, Proc. National Computer Conference, Vol. 48, 1979.