A Parallel Processor Operating System Comparison

Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

#CU-CS-085-75

December, 1975
(revised January, 1977)
(revised April, 1977)

This work was supported by the National Science Foundation under grant number MCS74-08328 A01

Index Terms: Operating systems comparison, parallel processors, simulation, performance evaluation, associative processors, array processors.

ABSTRACT

Three different operating system strategies for a parallel processor computer system are compared, and the most effective strategy for given job loads is determined. The three strategies compare uniprogramming versus multiprogramming and distributed operating systems versus dedicated processor operating systems. The level of evaluation includes I/O operations, resource allocation, and interprocess communication. The results apply to architectures where jobs may be scheduled to processors on the basis of processor availability, memory availability, and the availability of one other resource used by all jobs.

INTRODUCTION

The increasing complexity of computations in modern computing systems has led to the use of techniques for decomposing these computations into a set of parallel tasks to be executed on multiple processor computers, [1]. One popular architecture for executing certain kinds of parallel tasks is the single-instruction-stream-multiple-data-stream (SIMD) approach, and another approach is through the use of multiple, identical processing units, [2]. It is possible to combine the two approaches to obtain an architecture for executing several SIMD programs in parallel. In this paper, three operating system organizations for a combination parallel processor are compared with respect to their relative implementation complexities and effectiveness. Many of the comparisons made between the strategies transfer to other multiple processor organizations.

SIMD processors are characterized by the block diagram of Figure 1. The host processor is a conventional uniprocessor system, (a Burroughs B6700 in the case of the Illiac IV, [3], and a PDP-11 in the case of the Goodyear STARAN, [4]). The purpose of the host processor is to execute the operating system and other system software; the SIMD processor is essentially a functional unit of the host processor. The host loads the instruction memory with a SIMD program, and loads the processing element memories (PEMs) with data. The control unit (CU) then fetches and decodes instructions sequentially from the instruction memory and broadcasts each decoded instruction to an array of processing elements (PEs) which execute the given instruction (in parallel) on data stored in the respective PEMs. Parallel computation takes place in the PEs on the multiple data streams in a tightly coupled fashion.

SIMD processors are special purpose in the sense that a computation must be arranged in a manner such that a single instruction can be simultaneously applied to multiple data elements. Thurber and Wald list a set of suggested application areas ranging from air traffic control problems to solutions to differential equations, [5]. Arranging a computation so that it makes effective use of a SIMD organization can be a nontrivial task, (e.g. see reference [6]).

Multiprocessor organizations allow much more freedom in synchronizing the individual parallel computations, ranging from the organiza-

tion used in the Control Data 6600 where individual units are allowed to compute machine-level functions, [7], to complete isolation of one process from another as might be used in loosely coupled system. In the case of complete isolation, the parallelism of the machine takes place at a job level rather than at a process or task level, i.e., jobs are executed in parallel, not computations within a process.

The multiple control unit SIMD combination, hereafter MSIMD, allows each process to be decomposed into parallel SIMD-type tasks while still providing parallelism at a more loosely coupled level between control units. The range of synchronization times between control units can vary from procedure calls up to complete isolation. There is a loose coupling between control units and a tight coupling between processing elements. This combination of MIMD and SIMD architectures is not new in this paper; the original ILLIAC IV design [3] employed 4 control units. More recently, Radoy and Lipovski have also used the combination in their switched multiple instruction, multiple data stream architecture, [8].

The MSIMD organization allows one to make more effective use of the set of processing elements than a SIMD organization. Since the set of processing elements is shared by the multiple control units, an operating system can control the set of active jobs such that some control units will require large numbers of processing elements while other jobs require relatively few processing elements. This approach broadens the scope of jobs that the machine can execute efficiently, since it is not necessary for all jobs to simultaneously require a large number of data streams at any given time. The number of processing elements required by a job on a control unit corresponds to the number of data streams. However, an MSIMD architecture is still special purpose since all jobs are assumed to require multiple data streams during some point in their execution; the organization cannot effectively take the place of a general purpose computer for all types of programs. Although this approach will make processing element utilization more effective and user programming easier, it is not without its shortcomings. The facility for broadcasting information from each control unit to its allocated set of processing elements is more complex, and fast methods of loading/unloading the processing element

memories are not known, (as they are in the ILLIAC IV, [3]). The problem of information broadcasting has been considered and a design for a MSIMD architecture has been proposed, [9]; the I/O problem has not been solved to the degree that it has been in current SIMD machines.

An important asset of the MSIMD architecture over the SIMD architecture is the ability to execute system software directly on the machine rather than using a host processor. The machine becomes a separate entity rather than an extension to a sequential processor. System software, and in particular the operating system, can be written to take advantage of the (M)SIMD organization. Although an operating system is likely to be written as a sequential (SISD) program for most tasks, there are some phases that can be implemented as parallel algorithms, e.g. a preemptive scheduling algorithm can be written to also advantage of the SIMD organization, [10]; deadlock detection and avoidance algorithms illustrate inherent parallelism; most table searching operations can be done in parallel; etc. The number of processing elements required by the operating system during its parallel computation phases would correspond to the number of processes in the system in the first two examples, and to the number of table entries in the third example. For sequential operation, only one processing element is required.

In the following, three strategies for operating systems that execute on a MSIMD machine are considered. The simplest strategy is for one control unit to execute the operating system while the remaining control units operate as conventional SIMD machines sharing processing elements. The second strategy again uses a dedicated control unit for the operating system, but the remaining control units are multiprogrammed. The third strategy is for all control units to be multiprogrammed, with the operating system processes being executed on any control unit that requires system service, i.e., a distributed operating system. The dedicated uniprogrammed system is the simplest to construct, but has the disadvantage of no control unit sharing and the overhead of one control unit. The dedicated multiprogrammed solution is more difficult to implement, and has exorbitant control unit overhead. The distributed system utilizes the control units more effectively at the cost of complexity.

In the next section, a more specific environment for comparing the three operating systems is discussed, and the components that enter into the comparison are given. Next, the three strategies are described in more detail and then the results of the experiment are provided.

THE ENVIRONMENT OF THE OPERATING SYSTEM

The context of the experiment is a hypothetical MSIMD machine called the Multi Associative Processor (MAP), [11]. The system is composed of eight control units, (CUs), and 1024 processing elements, (PEs). Figure 2 illustrates the organization of the MAP system, where the individual units of the machine correspond to those discussed in the introductory section with the following exceptions. The host processor is replaced by an Input/Output system composed of channels, devices, and controllers. An I/O transaction takes place when a CU invokes a channel with an appropriate channel program. The instruction memory is replaced by eight main memory (MM) modules. Instructions and data for each SIMD program are stored in MM. Each CU has a preferred MM module, meaning that the CU can access that module without using a shared memory bus; the CU can access any other MM module via the shared memory bus, although bus conflicts may arise if multiple CUs attempt to simultaneously access modules other than their preferred module. Program sharing and message passing are permitted, but indiscriminant memory sharing is discouraged by the memory access oranization. The broadcast switch is used to route information from each CU to the set of PEs currently allocated to that CU, and to route information among the set of PEs allocated to a common CU. PE sharing among processes is not allowed at the hardware level, but is implemented by the software. Details of the memory organization and the broadcast switch are discussed elsewhere, [12], and are not reflected by the model used to compare MAP operating system strategies. For example, the model requires that each job allocated to a control unit also be allocated memory only in the preferred module for the CU. The possibility of a program overlapping MM modules was ignored since the model is not sufficiently detailed to represent memory bus conflicts.

In order for a job to be executed, all instructions must be loaded in MM, PEs must be allocated to the CU on which the job is scheduled, and the PE memories must be loaded with the multiple data streams. Once loading has taken place, the process corresponding to the job can compete for the CU if it is multiprogrammed, or it will have been allocated the CU in the uniprogramming case. All I/O operations take place through channels in the I/O subsystem, and the process requesting the operation is blocked during that operation. This ignores the possibility of I/O-compute overlap accomplished by buffering; it also ignores file organizations and interrupt handling. Channel queues are modeled, and delays due to busy channels are reflected in the results. The approach is realistic if random access files are assumed; otherwise, the simplified strategy will tend to favor multiprogrammed organizations over a uniprogrammed system since the latter strategy will idle a CU for the duration of an I/O operation. A more detailed study would include buffering.

There are two forms of process communication incorporated into the MAP system: passive and active communication. Passive communication, called synchronization, represents message passing; the sending process generates a message and passes it to the CU logically executing the receiving process. If the receiving process is not currently physically running on that CU, the sender is blocked until the receiver becomes physically active, and then both processes continue execution, i.e., the sender is blocked until the receiver is physically running and can receive the message. Active communication, called preemption, represents a higher priority process temporarily preempting the PEs allocated to a given process on a given CU. The effect is that the preempted process is blocked and its PEs are used by the preempting process; the preempting process temporarily deallocates a resource from a lower priority process for its own use. In particular, when process p_i preempts process p_j, p_i issues commands directed toward p_j. If p_j is not currently active, then p_i is temporarily blocked until p_j can receive the preemption commands. Once p_j receives the preemption commands, then p_j and the CU executing p_j become dormant while p_i uses the PEs allocated to p_j. At the conclusion of the preemption period, p_i and p_j resume operation on their respective CUs.

In this environment, a job is allocated to a CU whenever sufficient main memory and PEs are available. (The specific scheduling algorithms are discussed in the next section.) However, memory is allocated to a process (job in execution), while PEs are allocated to the CU and then shared among the set of processes that use that CU. The PEs are multiplexed and the PE memories are divided among the set of processes that share the CU allocated to the processes, i.e. PE memory blocks are allocated to processes rather than CUs.

The assumptions made about the hardware environment of the operating system do not necessarily restrict the results to the MAP system. A similar environment might exist in a multiprocessor system which incorporates eight processors, and which schedules jobs (SIMD or SISD) to those processors on the basis of the availability of two distinct resource types corresponding to main memory and PEs. The PE-type resource must be allocated to a processor and then multiplexed among the set of processes that share that processor. One example of such a system might be a multiprocessor that allocates jobs to processors on the basis of memory and disk drive availability, where the drive is allocated exclusively to a given processor and then shared among processes executed on that processor. Another example is a multiprocessor system with virtual memory. The MM resource corresponds to the address space allocated to each process, and the PE resource corresponds to the number of page frames allocated to a given multiprogrammed processor. Page frames in the physical memory are multiplexed among the set of processes that share a given processor.

THE OPERATING SYSTEM STRATEGIES

The basic considerations for the MAP operating system are whether or not multiprogramming should be incorporated, and whether or not the operating system should be distributed. Since a distributed operating system without multiprogramming is not a plausible consideration, that leaves the three strategies mentioned in the introduction as possibilities.

Examples of distributed and centralized operating systems for multiprocessors currently exist. Fabry describes a "control processor"

and "problem processors" in the PRIME system, [13], used to centralize operating system computations. Similarly, the Control Data 6000 series computers employ a single peripheral processor to execute the primary portions of the operating system, with some operating system code being executed on the central processor. On the other hand, Multics [14] and HYDRA [15] both support multiple processors by distributing the operating system over all available processors, each process calling the operating system as needed. Gonzalez and Ramamoorthy consider the execution time of parallel programs under centralized and decentralized strategies and find that decentralized control favors their environment, [16]. In this study, the three strategies are compared in a MAP environment, with the measures of quality being batch job turnaround time, throughput rates, and average input queue lengths.

The dedicated uniprogrammed operating system, called UDED, allows a maximum of seven processes to be in execution. Each process can have from one to approximately 1000 PEs allocated to it, with the constraint that the total number of allocated PEs for all CUs does not exceed 1024. This does not preclude a process from dynamically requesting and releasing PEs during its lifetime. The organization of the system includes an input queue for jobs to be executed and each job has an initial request for PEs. The job scheduling algorithm inspects the input queue in a first come, first served fashion, selecting the first job that can be allocated to a vacant CU, with the currently available PEs. In UDED, it is assumed that the job will not request more main memory than exists in a single memory module*. Once a job has been allocated a CU and PEs, then the corresponding process may request I/O, request or release PEs, synchronize with another process, or preempt another process. The idle time introduced by one of these actions leaves the CU inactive until the requested operation has completed.

The dedicated multiprogrammed operating system, called MDED, allows a maximum of four processes to compete for a given CU, with the operating system remaining on a uniprogrammed CU. The CU scheduling algorithm becomes considerably more complex since the processes that share a CU must also share the same main memory module and the same subset of PEs and their respective PEMs. The philosophy of the

* This constraint does not hold for the MAP design, but is used to simplify the model.

scheduling algorithm is that PEs are a more valuable commodity than main memory, thus a variant of a best fit algorithm is employed with respect to PEs. CU Deallocation is described as follows:

Let R_i = number of PEs currently allocated to CU_i, $(1 \leq i \leq 8)$

M_i = number of main memory locations in MM_i allocated to CU_i, i.e., in use

$\overline{R}$ = number of PEs configured into the machine

$\overline{M}$ = number of main memory locations in each module

r_{ij} = number of PEs required by process j on CU_i $(1 \leq j \leq 4)$*

m_{ij} = number of main memory locations in MM_i allocated to process j

Then,

$$R_i = \max_{1 \leq j \leq 4} (r_{ij})$$

$$M_i = \sum_{j=1}^{4} m_{ij}$$

Suppose process j completes execution on CU_i, then

1. CU_i can service a new job
2. $M_i \leftarrow M_i - m_{ij}$
3. $\hat{R}_i \leftarrow \max_{\substack{1 \leq k \leq 4 \\ k \neq j}} (r_{ik})$;

   if $\hat{R}_i < R_i$ then begin deallocate $(R_i - \hat{R}_i)$ PEs from CU_i; $R_i \leftarrow \hat{R}_i$

   end

That is, the number of PEs allocated to CU_i is the maximum number of PEs allocated to any process executing on CU_i. If CU_i has less than 4 processes currently executing, then:

---

* Processes on CU_i are numbered 1 to 4; when process j leaves the system, the new process assigned to the CU takes the index of the completed process.

Choose a process, s, from the input queue such that s requests r_s PEs and m_s main memory location where

r_s is the largest PE request such that $r_s \leq R_i$ and $m_s \leq \overline{M} - M_i$;

process s is allocated the CU vacancy, i.e. process s is chosen as the best fit with respect to the number of PEs allocated to the CU.

If no such process s exists (i.e. best fit is not possible), then the following actions take place:

1. Choose process s such that $\min_s(r_s)$ and $m_s \leq \overline{M} - M_i$
2. if $r_s - R_i < \overline{R} - \sum_{k=1}^{8} R_k$ then
   begin allocate $(r_s - R_i)$ PEs to CU_i;
   $R_i \leftarrow r_s$; load and activate process s
   end

When no best fit is possible, the job which requires the fewest additional PEs be allocated to the CU is chosen. The reason for choosing this job is that the other jobs already allocated to the CU do not require the additional PEs; hence whenever they are using the CU, the additional PEs are idle. The algorithm attempts to keep jobs that require similar numbers of PEs on the same control unit. It also tends to keep the number of PEs allocated to a CU static. The algorithm initially attempts to run jobs requesting few PEs, but as the system continues to run, some CUs begin to execute those jobs with larger PE requirements.

The four processes that reside on the CU are serviced by a round robin scheduler, where the time quantum may be used for preemption, or optionally, scheduling only takes place when the job currently running becomes blocked.

The distributed multiprogramming operating system, called MDIS, employs the same schduling philosophy as MDED. The difference is that the operating system does not use any of the multiprogramming levels assigned to the CU, and the operating system code is either distributed over the eight memory modules or else stored in a ninth memory

module not shown in Figure 2. Additionally, the operating system processes have higher priority than any of the user processes on the CUs. A supervisor call blocks the calling process and activates the operating system on the same CU. Note that when two or more CUs request the operating system, then the code is being shared among the CUs. Although critical sections must be handled, the only adverse effect is that the probability of memory conflicts increases. An unresolved problem with the MDIS that does not occur with either MDED or UDED is interrupt handling. The particular CU that should receive an interrupt is not defined in this study.

The three operating system strategies do not exhaust all possibilities for a design--they only represent three broad classes. There are a variety of other scheduling strategies that could be tested, but for the level of comparison discussed here it is felt that a proper perspective of relative performances can be obtained.

THE COMPARISON METHOD AND MEASURES

The comparison of the performance of the three operating systems is accomplished through simulation. A model has been constructed to exercise the MAP system under the control of the different strategies using the same load. The system parameters to the model are listed in Table 1, and the job load description parameters are listed in Table 2. All time units are one hundredths of seconds. A single program has been written so that the operating system being modeled is determined by system parameters 1, 2, and 9; the same simulation code is used to test all three options. The remaining parameters have been chosen to allow the simulation program to represent various memory sizes, I/O rates, PE allocation times, and synchronization times.

The round robin time slicing algorithm is used in all cases, where a time quantum of zero indicates that a process relinquishes the CU only if it requests an I/O, PE allocation/deallocation, synchronization, or preemption. (UDED should always be executed with a time quantum of zero, otherwise at time quantum expiration the model would deallocate the CU from a process and then immediately reallocate it to the same process for another nonzero time quantum.) The I/O operation

time is determined to be $a\lceil y/x \rceil + b$ where x is the number of words in a mass storage physical record unit, a is a transfer rate, b is a latency time, and y is the number of words to be transferred by the operation. Since the job scheduling algorithm for UDED is a degenerate case of the job scheduling algorithm for both MDED and MDIS, the same scheduling algorithm is used in all three systems. The amount of time required to accomplish dynamic PE allocation and deallocation is determined as follows: Deallocation and allocation of n PEs required cn+d time units, where c and d are input parameters. If an insufficient number of PEs are currently available for an allocation request, the request is queued and the process is blocked until PEs become available. Operating system execution times (items 9 in Table 1) refer to the amount of time required for the distributed system to carry out the given function on the requesting CU; e.g., if a process requests an I/O in MDIS, parameter 9c describes the amount of time for which that CU is being used by the operating system to set-up the I/O operation and the CU cannot be scheduled to another prcess during the given time period.

The job load is described by the distributions listed in Table 2. Each distribution is described by defining a mean value, a standard deviation, and whether the distribution is normal or of the exponential family. If an exponential-type distribution is specified with the mean value equal to the standard deviation, then the distribution type is exponential; if the mean value exceeds the standard deviation, then the distribution type is Erlang; if the mean value is less than the standard deviation, the type is hyperexponential. The job characteristics shown in Table 2 reflect the level of detail at which each job is modeled. (The effects of a job on the system are described by resource requirements, I/O operations, and the pattern of interprocess communications.) Input parameters for job load parameters 4-9 specify a distribution for the mean value of the given characteristic. For example, if the distribution type for parameter 4 is normal, then the distribution describes a normal distribution of mean time between I/O operations, and an individual job's inter I/O time distribution is normal with a mean value determined by sampling the parameter 4 distribution with a standard deviation of one fourth the mean value. Each job has a unique distribution generated from job load parameters 4-9.

The distributions derived from job load parameters 8 and 9 determine the time between synchronization and preemption operations, but

they do not determine the identity of receiver processes for these operations. Receiver processors are determined by randomly generating process identifications. If a process that is to receive a synchronization or preemption does not exist, then another process identification is generated. This can lead to early termination of the simulation of a system when only one process is active in the entire system and it is attempting to execute a synchronization or preemption. It also causes the amount of blockage due to process communication to increase with the number of processes currently active in the system. Although precautions have been taken to eliminate the possibility of deadlock on communication operations, this too can occasionally occur.

The most obvious comparison measures are the mean throughput rate and the mean turnaround time of jobs. It is also useful to determine the maximum and mean input queue lengths induced by a job load. Other comparison measures are the amount of CU blockage due to I/O requests, PE allocation, synchronization, and preemption. The simulation models provide these data as well as the number of jobs input, the number of jobs output, status of CUs at termination time, amount of CU time spent in operating system execution in the case of MDIS, channel utilization, and histograms of job allocation to CU time, CU blockage time, job active on a CU, I/O blockage, PE allocation blockage, synchronization blockage, and preemption blockage. In the next section, the results of testing various configurations and job load is discussed.

EXPERIMENTAL RESULTS

More than 75 different runs of the simulation model were made, and data from only a subset of those runs are presented here. The specific system parameter values used in the simulation runs were determined by the MAP architecture in most cases. The job load parameter values were chosen partially by observations of MAP programs, [9], and partially by the results of other simulation runs (e.g. in the case of interarrival times). The job load is, to a large degree, hypothetical since no real job load for MAP currently exists.

Stabilization Times

The first experiment was to determine the time required for the program to stabilize for each of the three systems, i.e. as the simulation is initiated, all CUs are idle, all resources are unallocated, and the input queue is empty. As jobs arrive in the system, the CUs begin operation and resources are allocated. The amount of time for the system to reach a state where the queue length and resource utilization tend toward a constant figure is referred to as the stabilization time. The job load was first described as having a mean job interarrival time of 3.0 seconds, each job required a mean CU computation time of 25.0 seconds, (the interarrival time was exponentially distributed, and the service time was normally distributed). The main memory configuration included 32K words per module and each job had a mean main memory request of 8K words (normally distributed with a standard deviation of 2K words). The MM configuration and request figures are hypothetical. PE interrequest times were normally distributed with a mean of 12.5 seconds, the number of PEs on each request determined by a normal distribution with mean value of 64 and a standard deviation of 32 PEs. PE request figures are based on actual MAP programs. The I/O request pattern is determined by a mean time between I/O operations of 2.5 seconds and a mean number of words on each transfer equal to 512. Mean time between synchronizations was 6.0 seconds and mean time between preemptions was 12.5 seconds, both exponentially distributed. Each system was then run for 300, 450, 600, and 750 seconds of simulated time. As might be expected, UDED had stabilized before 300 seconds, while MDED and MDIS both stabilized at some time greater than 300 seconds but less than 450 seconds. Most succeeding runs use a simulated time of 450 seconds of execution.

Maximum Arrival Rate Comparisons

The next experiment was to determine the maximum arrival rate each system could support, i.e., what is the maximum throughout rate for each system. In order to remove the effect of excessive synchonization and preemption due to the number of active processes, job load parameters 8 and 9 (in Table 2) were set so that the mean time between these activities was much larger than the expected CU service time, i.e., synchronization and preemption occur so infrequently that all

processes are essentially independent of one another. Although it was expected that the multiprogramming configurations would perform better than the uniprogram configuration, especially since I/O-compute overlap was disallowed, a comparison of relative performance was desired. The mix was tested with mean interarrival times (exponentially distributed) between 1.0 and 2.5 seconds in increments of 0.25 seconds. Figure 3 indicates the observed throughput rates for all three systems under the various arrival rates. UDED became saturated at a mean interarrival time between 1.75 and 2.00 seconds (about 0.55 jobs/second); MDED appears to reach saturation when the mean interarrival time is about 1.50 seconds (the corresponding arrival rate is about 0.67 jobs/second); MDIS begins to saturate at a mean interarrival time of 1.00 to 1.25 seconds, (the rate is about 0.80-1.0 jobs/second). Figure 4 describes the percentage of jobs that arrived in the system that were either served or were in service when the simulation halted. It is somewhat surprising to see that MDED resembles UDED more than it does MDIS in these results. To check the consistency of the observed phenomena, the model was again exercised with the same interarrival rates, but for 600 and 750 second runs. The results were the same for all three versions of the system. The conclusion is that once a system reaches a saturation load, performance declines rapidly, regardless of the configuration. In Figure 5, the maximum queue lengths and mean queue lengths for the input queue are plotted against the interarrival time for each system. Note that data points for UDED at 1.25-1.75 second interarrival times (and at 1.25-1.50 seconds for MDED) is determined primarily by the length of simulated time, since both systems are then experiencing an arrival rate greater than can be handled. Figure 6 illustrates the percentage of utilization of CUs allocated to user programs, (i.e., the fraction of the total time during which the CU was allocated to a job). UDED reaches over 90% utilization at 1.75 second interarrival times; MDED approximates 90% utilization at 1.75-1.50 second interarrival times, and MDIS becomes saturated between 1.25 and 1.50 second interarrival time.

The Effect of Synchronization

The next experiment employed the job mix described at the beginning of this section, i.e., synchronization and preemption times were allowed to affect the results. As discussed previously, the number of

logically active processes partially determines the frequency of synchronizations; when more processes coexist, communication requests are more frequent and each "conversation" results in longer periods of CU blockage due to the possibility of chains of CUs waiting for some process to become active. Even though circular waits are possible, they are avoided in the simulation whenever the model can easily detect them.* Figure 7 indicates the observed throughput rates for this experiment, using a simulated time of 450 seconds. For the mean interarrival times between 1.50 seconds and 2.50 seconds, the results are similar to the previous experiment. For smaller interarrival times, the effect of synchronization and preemption begins to override the throughput rate. For the higher rate, more processes simultaneously exist and CU blockage begins to drastically reduce the throughput rate for MDED and MDIS. UDED is not effected much, since there is a maximum of only 7 logically active processes and the communication blockage that exists is not severe. For example, at the 1.25 second interarrival time tests, the average amount of time a process is blocked due to both forms of communication is about 0.15 seconds in UDED, 11.25 seconds in MDED, and 12.55 seconds in MDIS. For this job load, the effective saturation load for each system is reached with mean interarrival times of 1.50, 1.75, and 2.0 seconds for MDIS, MDED, and UDED respectively. The amount of interprocess communication is a sensitive performance parameter, especially for the multiprogrammed operating systems. Any development of these classes of systems must pay careful attention to the policies employed in implementing interprocess communication; deadlock detection or avoidance cannot be ignored.

Operating System Overhead

The final phase of experimentation is concerned with the amount of operating system overhead with respect to CU utilization. Both UDED and MDED require a dedicated CU for implementing the operating system, and so the amount of the CU resource is easy to quantify; it is 12.5% of the total resource. In the experiments with MDIS, it is

---

* Deadlock occurred in only one run, and that was under a load with mean intersynchronization time of 3.0 seconds, mean interpreemption time of 7.0 seconds, after 300 seconds of simulated time.

assumed that the time to remove a process from a CU is 10 ms, the time to schedule a new process to a CU is 20 ms, and the time to initiate an I/O operation is 10 ms (in addition to the channel transfer time). PE allocation requires 20 ms of overhead plus 10 ms for each PE allocated or deallocated. Synchronization overhead time is ignored and preemption overhead time is 20 ms. MDIS was tested with normally distributed CU active times where the mean value ranged from 20.0 seconds to 35.0 seconds and the mean interarrival times of 1.25, 1.50, and 1.75 seconds. In all cases, the percent of the total simulated time ranged between 0.6 and 0.9%. The amount of CU time spent on operating system overhead is negligible, indicating that the use of a dedicated CU for the operating system is wasteful of that resource. The operating system does not require enough time to justify dedicating a CU for its exclusive use, although the complexity of the design may be more difficult.

CONCLUSION

The same simulation program was used to test all three versions of operating systems in this study. Input parameters to the model determined the configuration being tested. The results of the study indicate that for an environment that does not make excessive demands on the machine, the uniprogrammed approach with a dedicated CU for the operating system is an attractive solution. Actual UDED performance would be better than predicted by the model if I/O buffering were incorporated to allow I/O-compute overlap. As the job load increases to a point of saturation, the distributed, multiprogrammed operating system appears to have a performance edge over the dedicated multiprogrammed system that makes the added complexity worthwhile. There is little data to justify the implementation of MDED, since its complexity may approach that of MDIS, but its performance falls far short.

The comparative studies also indicate the importance of using a sound policy for process communication in a system simultaneously supporting a large number of processes. As the number of processes grows, the performance of the system may be degraded by processes involved in busy waits or other forms of CU blockage; deadlock becomes important to consider in these cases.

Although the model was derived to represent the MAP computer system, it could equally well be applied to any multiprocessor system where processor scheduling is determined by processor availability, memory availability, and the availability of at least one other system resource by each job. The level of modeling does not make the array processor architecture a primary point other than that PEs are the "other system resource". The SIMD portion of the architecture does not place additional constraints on the simulation.

In any simulation study, validation of the model is an important consideration. There is really no way to validate the model by analytic techniques, since it is too detailed. It is also not possible to validate by measurement since no real hardware for MAP exists.

An important lesson that was learned from the work was that, at least in simulation, multiprogramming is more difficult to implement than distribution. Upgrading the program from UDED to MDED required approximately 2 man months, while incorporating MDIS into MDED required only an additional week or two. Since these modification problems reflect actual operating system implementation considerations, it appears that software development for real operating systems would require the same proportions of effort.

ACKNOWLEDGEMENT

The author wishes to thank the National Science Foundation for the support of this work under Grant number MCS74-08328 A01.

REFERENCES

[1] J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing" ACM Computing Surveys, Vol. 5, No. 1, pp. 31-80, March, 1973.

[2] M. J. Flynn, "Some Computer Organizations and their Effectiveness", IEEE Transactions on Computers, Vol. C-21, No. 9, pp. 948-960, September, 1972.

[3] G. H. Barnes, et al, "The Illiac IV Computer", IEEE Transactions on Computers, Vol. C-17, No. 8, pp. 746-757, August, 1968.

[4] E. W. Davis, "STARAN Parallel Processor System Software", in AFIPS Proceedings of the NCC, Vol. 43, pp. 17-22, 1974.

[5] K. J. Thurber and L. D. Wald, "Associative and Parallel Processors", ACM Computing Surveys, Vol. 7, No. 4, pp. 215-255, December, 1975.

[6] H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", Journal of the ACM, Vol. 20, No. 1, pp. 27-38, January, 1973.

[7] J. E. Thornton, Design of a Computer: The Control Data 6600, Scott Foresman and Company, 1970.

[8] C. H. Radoy and G. J. Lipovski, "Switched Multiple Instruction, Multiple Data Stream Processing", in Proceedings of the 2nd Annual Symposium on Computer Architecture, 1974, pp. 183-187.

[9] R. D. Arnold and G. J. Nutt, "The Architecture of a Multi Associative Processor", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-070-75, 50 pages, June, 1975.

[10] G. J. Nutt, "Sample Programs for a Hypothetical Computer", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-058-74, 26 pages, October, 1974.

[11] G. J. Nutt, "A Parallel Processor for Evaluation Studies", in AFIPS Proceedings of the NCC, Vol. 45, pp. 769-775, 1976.

[12] G. J. Nutt, "Memory and Bus Conflict in an Array Processor", to appear in IEEE Transactions on Computers.

[13] R. S. Fabry, "Dynamic Verification of Operating System Decisions", Communications of the ACM, Vol. 16, No. 11, pp. 659-668, November, 1973.

[14] E. I. Organick, The Multics System: An Examination of its Structure, MIT Press, Cambridge, Massachusetts, 1972.

[15] W. Wulf, et al., "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol. 17, No. 6, pp. 337-345, June, 1974.

[16] M. J. Gonzalez, Jr. and C. V. Ramamoorthy, "Parallel Task Execution in a Decentralized System", IEEE Transactions on Computers, Vol. C-21, No. 12, pp. 1310-1322, December, 1972.

Figures

Tables

	Description	Range of Values		
		UDED	MDED	MDIS
1.	Number of user CUs	7	7	8
2.	Number of levels of multiprogramming/CU	1	4	4
3.	CU time slice*	-	arbitrary	arbitrary
4.	Main memory module size	arbitrary	arbitrary	arbitrary
5.	Factors determining time to do I/O**	arbitrary	arbitrary	arbitrary
6.	Number of I/O Channels	1-8	1-8	1-8
7.	Factors determing time to accomplish PE allocation	arbitrary	arbitrary	arbitrary
8.	Mean preemption time (Uniform distribution)	arbitrary	arbitrary	arbitrary
9.	Operating System execution time to			
	a. deallocate a CU	0	0	≥ 1
	b. schedule job to CU	0	0	≥ 1
	c. initiate I/O	0	0	≥ 1

* Time slicing is not used in UDED. A value of zero indicates that there is no time quantum used in MDED and MDIS; a process relinquishes the CU when it is blocked.

** I/O time is determined to be $a\lceil y/x \rceil + b$, where a, b, and x are input parameters, and y is the number of words to be read/written in a given operation.

SYSTEM PARAMETERS

Table 1

Description

1. Mean, standard deviation of hyperexponential or Erlang distribution describing job interarrival time.

2. Mean, standard deviation of normal, Erlang, or hyperexponential distribution describing job processing time.

3. Mean, standard deviation of normal, Erlang, or hyperexponential distribution describing job main memory requirements.

The next 6 distributions describe the mean and standard deviation of means for distributions of individual jobs. The standard deviation of each is 25% of the mean used for the particular job. Each class is either normal or Erlang, determined by input parameters.

4. Distribution class for time between I/O operations.

5. Distribution class for number of words in I/O operation.

6. Distribution class for time between PE allocations.

7. Distribution class for number of PEs currently allocated to a CU (always a normal distribution; initial value for any job is mean + 2 * (standard deviation)).

8. Distribution class for time between synchronizations.

9. Distribution class for time between preemptions.

JOB LOAD PARAMETERS

Table 2

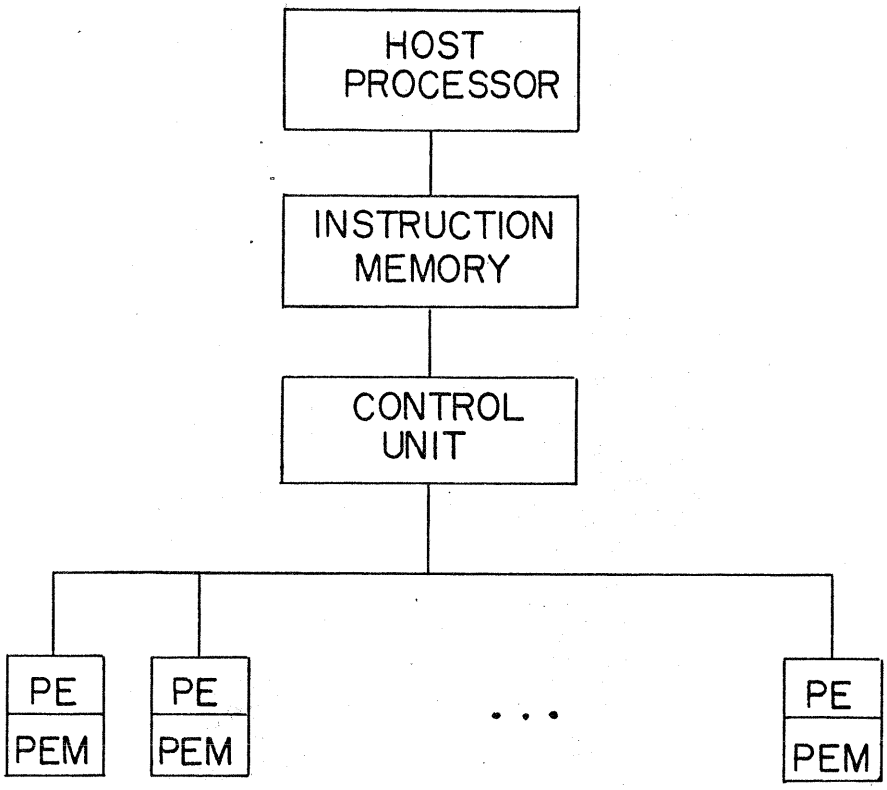
HOST PROCESSOR
INSTRUCTION MEMORY
CONTROL UNIT
PE
PEM
PE
PEM
. . .
PE
PEM

FIGURE 1

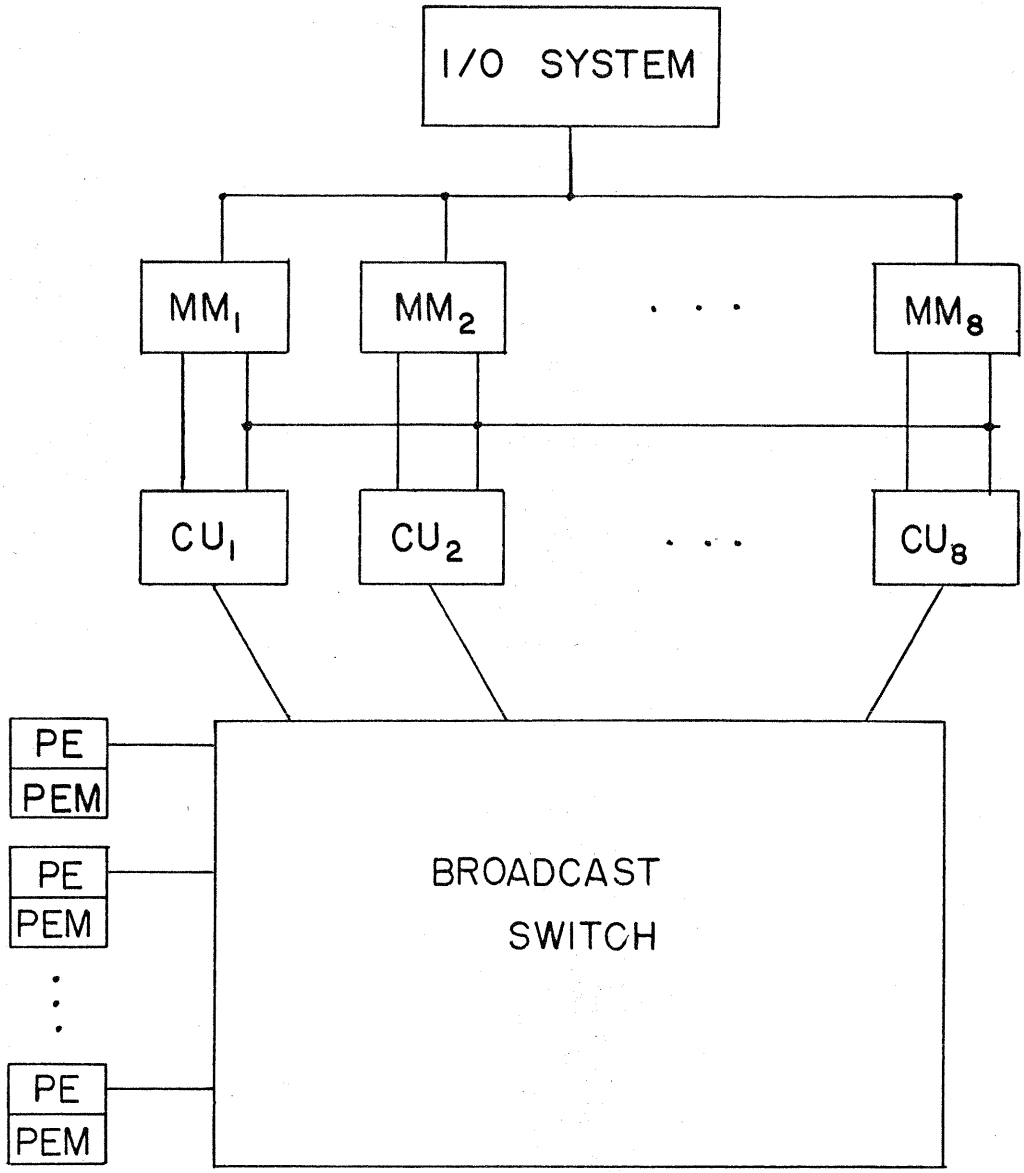
I/O SYSTEM
MM1
MM2
MM8
CU1
CU2
CU8
PE
PEM
PE
PEM
PE
PEM
BROADCAST
SWITCH

FIGURE 2

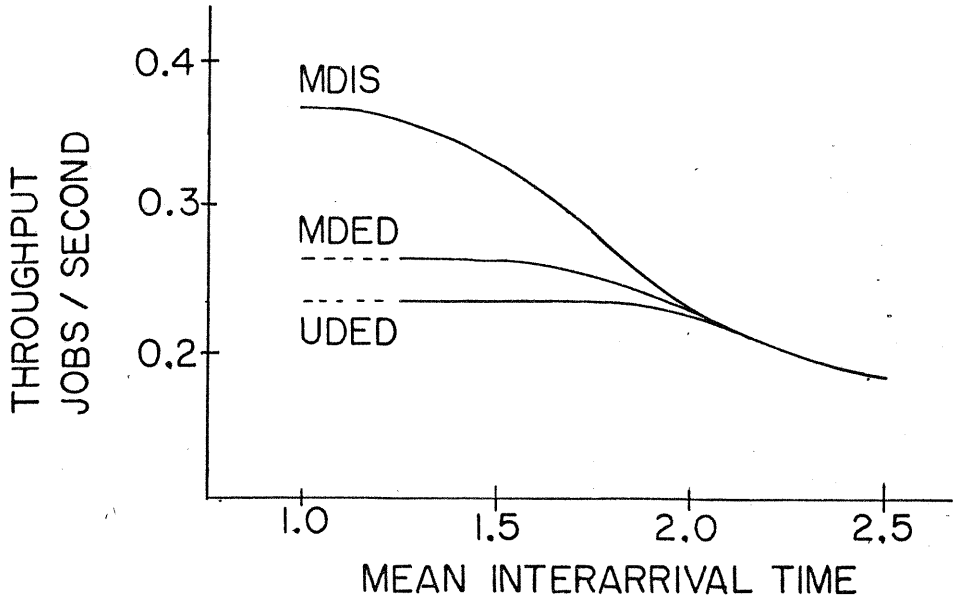

FIGURE 3

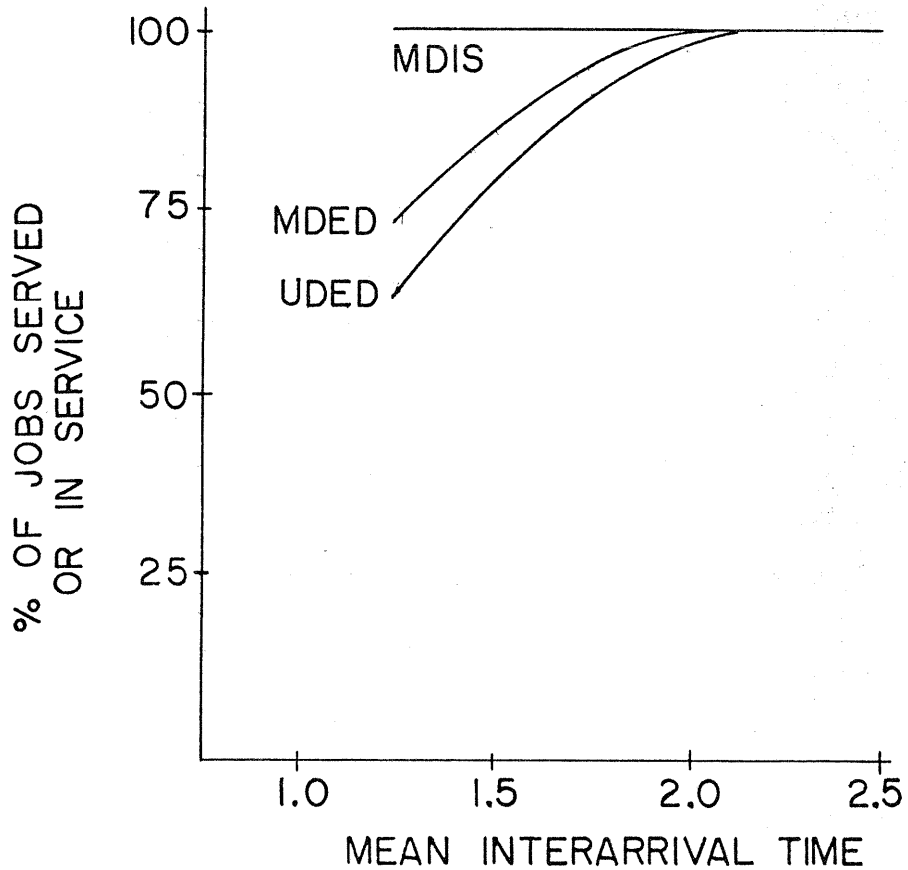

FIGURE 4

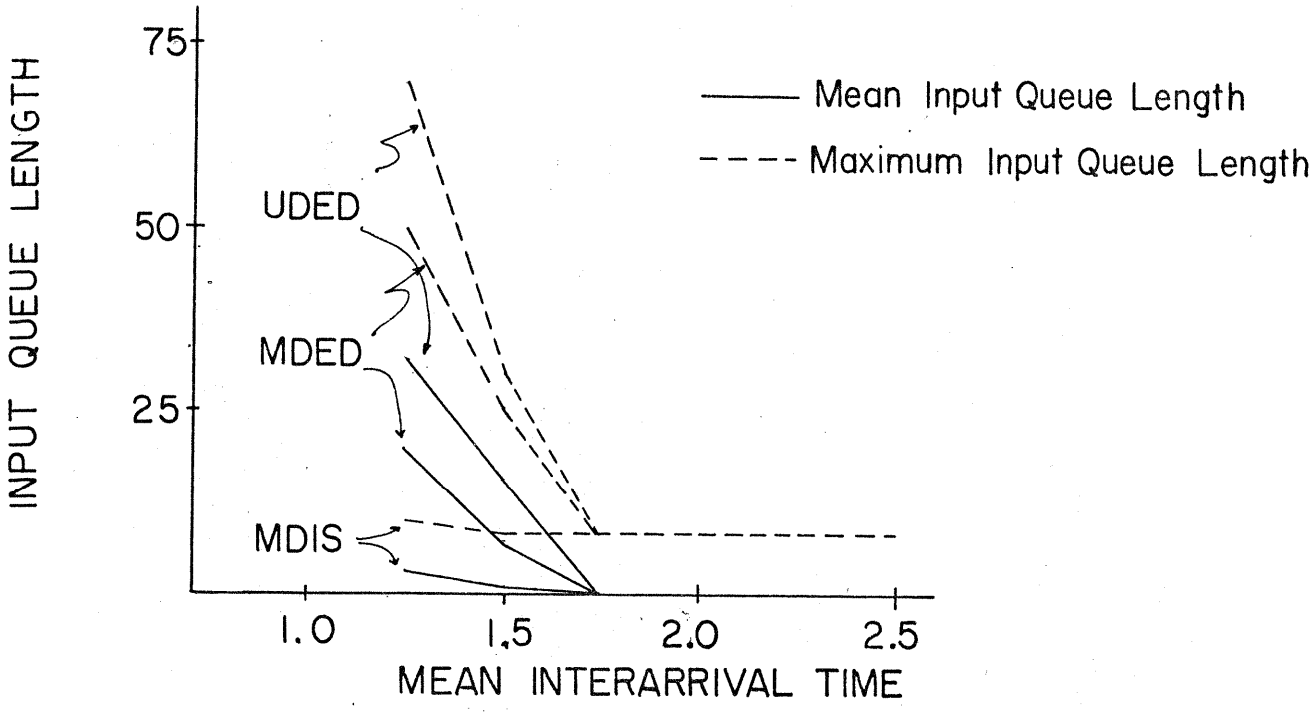

FIGURE 5

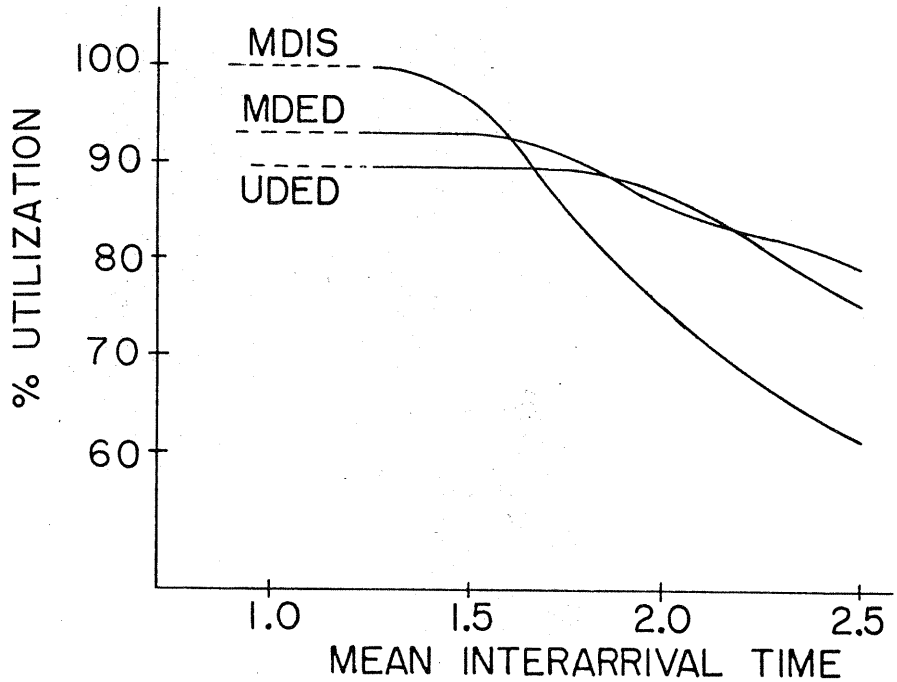

FIGURE 6

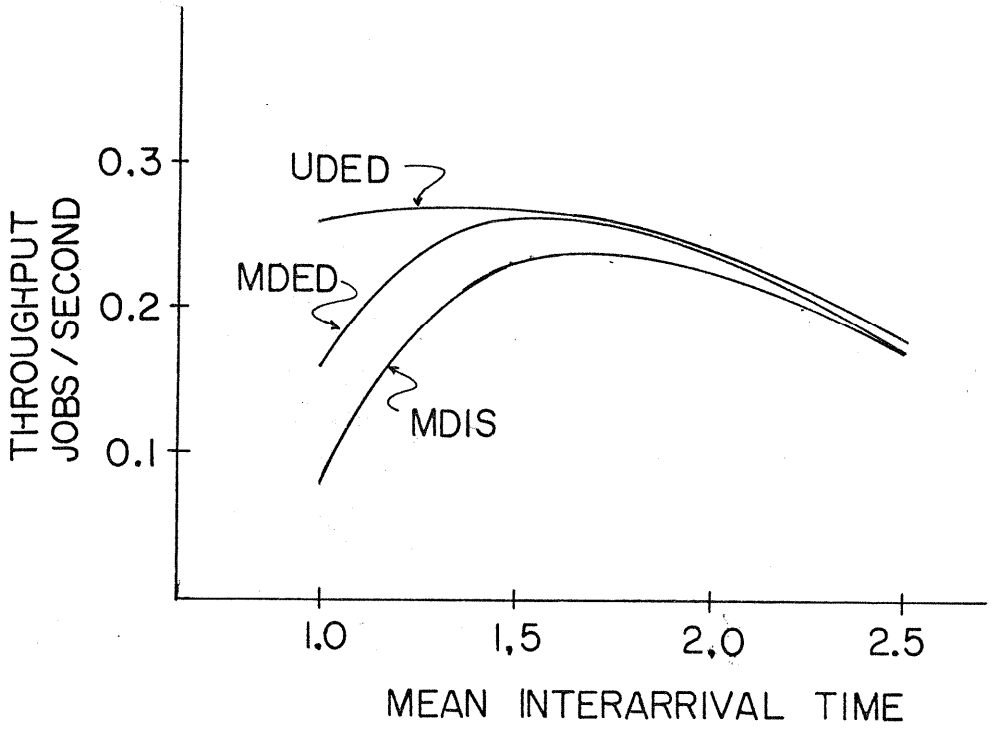
THROUGHPUT
JOBS / SECOND
0.3
0.2
0.1
UDED
MDED
MDIS
1.0
1.5
2.0
2.5
MEAN INTERARRIVAL TIME

FIGURE 7