A MULTI-MICROPROCESSOR COMPUTER SYSTEM ARCHITECTURE*

Bruce W. Arden and Alan D. Berenbaum
Princeton University

The development of microprocessors has suggested the design of distributed processing and multiprocessing computer architectures. A computer system design incorporating these ideas is proposed, along with its impact on memory management and process control aspects of the system's operating system. The key design feature is to identify system processes with microprocessors and interconnect them in a hierarchy constructed to minimize intercommunication requirements.

Key words: computer networks, distributed processing, microprocessors, multiprocessing, parallel processing

CR categories: 4.32, 4.35, 6.21

Introduction. In the near future, computer system designers are expected to bring more computer power to more people than is currently feasible. At the same time, it is reasonable for the computer system designers to expect the computer architects to produce hardware structures incorporating system algorithms of greater complexity so that more of the general instructions go to serving the users. In the past, the architect has delivered more computing to the user by constructing bigger and faster central processors, possibly connecting two or three CPUs together. This approach, though, has its limitations, in cost, complexity and reliability. Instead of large, centralized processing systems, an increasingly attractive alternative is to employ multiple less-expensive processors.

Present Systems. The simplest approach to distributing computing power is to set up independent minicomputers systems wherever there is a demand, as has been done in many universities and corporations. Although straightforward and inexpensive, this approach is limited with respect to information and resource sharing. Any common database is obviously difficult to implement and sharing of underutilized resources is limited. The problem of sharing is approached by networking. Examples of such systems include UC Irvine's Distributed Computer System [6] and the ARPA net [18]. However, the restrictions on interprocessor communication, necessary for a far-flung system like ARPA, may be unnecessarily severe for a more centralized system.

It is possible to configure a multiple processor system in a more "intimate" manner than a loosely-coupled network. If the computing system enables programs, which can be separated into parallel segments, to be executed in parallel efficiently, it is possible for a collection of small, low-speed processors to do the work of one high-speed processor. This is the motivation behind Carnegie-Mellon's C.mmp system [21]. With this system the designers claim a set of DEC PDP-11's can do the work of an IBM 370/158 at a substantially lower cost [13]. Alternatively, tightly-coupled multi-miniprocessor systems can be used to effect a modular expansion of computer power. This technique has been employed for Bolt, Baranek and Newman's PLURIBUS system [8].

Although it permits a much higher degree of parallelism, C.mmp is similar to systems like MULTICS and those running on a 370/158. Despite their relative low cost, the PDP-11 processors on C.mmp are a valuable resource like the CPU of a 370/158, and so must be multiprogrammed in the interests of efficiency. The problems of process scheduling and resource allocation must still be handled in a centralized manner, at least in so far as only one set of system tables must be maintained [22]. By taking advantage of rapidly advancing technology and altering the system design philosophy, it should be possible to

*Research supported by NSF Grant No. DCR74-18655.

construct a system that is at least as powerful, is highly modular in construction, yet permits the use of much simpler operating system structures. The system described in the following sections is an attempt to meet these criteria, especially in the environment of a large, central time-sharing facility with shared data base.

Microprocessor Potential. In a few years, microprocessors and semiconductor memories will become larger, faster and cheaper [20]. As block-oriented random access memories are developed, extremely large, if relatively slow, central memories are possible. As chips will still be constrained by the number of pins available, the complexity of a system will come not in the logical design, but in the interconnection of actibe components.

A microprocessor cannot have both the computational power of a larger machine along with its input-output flexibility. However, it is possible to program special purpose microprocessors which can handle som e aspect of resource manipulation. These processors can be tied together to form a distributed function processing complex [15], which has a broader capability than any of its parts. Alternatively, one can consider operating system functions that were formerly implemented in software to be embodied in the hardware of the special-function processors.

Memory Potential. It has been felt that reductions in operating system complexity can be achieved through the use of very large central memories [11]. With the type of semiconductor central memory used now (MOS main store with a bipolar cache) it is unlikely that a central memory will grow more than an order of magnitude in size, so that a typical large system might have 10 or 20 megabytes of main store. However, it is unlikely that the need for sophisticated memory management techniques will vanish at that point, since increased processor technology (plus increased user sophistication) will probably increase demand on main memory at the same rate the memory size increases. It appears that a different type of organization will be necessary to achieve much larger memory sizes. Contemporary time-sharing computers use a drum to simulate a large random access memory. Future mini-mass memories, fabricated out of CCD's, for example, will be larger than contemporary drums, at similar prices but with lower access times. These memories are, of course, restricted in that data are organized in rings and an individual byte of memory may not be directly accessible. However, if the memory system is hierarchical, this may not be a disadvantage. If a processor accesses memory only through a local memory or cache, then the main memory never need transfer data in groups smaller than a cache block. If the ring size of main memory is the same as a cache block, and the transfer logic is sufficiently intelligent to start a transfer in the middle of a block, then information can be transferred between memory levels without rotational delay. There are tradeoffs in the design of a block size. The larger the wordsize, the shorter the ring size, and hence, the faster the transfer rate at the cost of greater interconnection complexity. Likewise, a small blocksize leads to increased transfer rate, at the cost of greater memory management overhead [7].

Design concepts. The availability of sophisticated microprocessors suggests the design of inexpensive multiprocessor systems [16]. It is proposed here that given a network composed of connection-limited, inexpensive microprocessors, the design challenge is to find a modular organization for a general system which permits the individual parts to operate quite independently with a minimum of overhead interference of the kind present in current multiprocessor systems. One approach is to identify the various processes in a process-structured system and implement each as a separate microprocessor. When a process included a critical region only a single microprocessor would be used (e.g., storage and device allocation). If more than one instance of a process were permitted (e.g., drivers for multiple printers), then several microprocessors dedicated to that function would be included. Extending this construction to user-generated processes, as opposed to system processes, a single microprocessor would be dedicated to each user process. Since a process is never swapped and a process identifier is equivalent to the processor identifier, there is no need for system-wide process queues. However, the question arises whether microprocessors dedicated to a process for the lifetime of the process is the best approach. Considering the cost of microprocessors, the possibly large idle periods (or "busy waiting") are not strong counterarguments but, the increasing interconnection complexity with large numbers of microprocessors is a difficulty. This question is considered in more detail with the aid of a simple queueing model later on.

The large number of processors available for assignment to processes is also suitable for parallel processing applications. Process creation, termination and synchronization, although simple, are too time consuming with respect to primitive arithmetic operations to have concurrency at the arithmetic expression level. The parallelism would be at the more macroscopic level of processes. By structuring a large program as a collection of independent processes, it should be possible for many large programs to execute in a reasonable time on relatively slow processors.

The identification of processes and processors is conceptually attractive and reduces, to some extent, process interaction because process state is maintained in a hardware device and need not be transferred to a process state table. However, the critical problem is how to make a user process execute for relatively long periods of time without the need for other system services. Stated differently, if the memory associated with a user microprocessor is regarded as a cache or buffer store, the "hit ratio" or references to buffer addresses compared to external addresses must be very large. This much analyzed statistic can be increased by having a large buffer store, possibly relatively slow, or by increasing the fraction of the buffer store dedicated to data. The latter can be done by storing the instructions in a succinct, high level language form (e.g., APL) and interpreting them by means of microinstructions obtained from a ROM. The design that follows is a preliminary version of a general system based on microprocessor implemented system and user processes with limited processor interconnections.

Processor description. The basic service microprocessor, called a program processor, would be a $50 15-bit microprocessor attached to a high-speed buffer memory, a read-only control memory and a request bus connected to service processors. It has been shown that a small cache memory can effectively increase the speed of a larger, slower memory [10], so it is likely that a relatively small buffer memory will be sufficient for maintaining the processing rate of a program processor. If the system contains a large amount of central memory, there will be little need for paging to the file system, the third level in the memory hierarchy. As block-oriented memory becomes inexpensive, it will be economically feasible to maintain at least 1M byte of memory per process in the second level of central memory. This is as large a memory as that provided for a single process in many current, large scale multiprogramming systems.

As has been mentioned above, the critical problem in a large system employing a variety of small parts is that of interconnection. For a large system it would not be unreasonable to expect 100 program processors executing at once. It becomes difficult to connect those 100 processors to the various resources of a large computer system. It is therefore necessary to cluster program processors around a central service facility. A cluster is illustrated in figure 1. The whole cluster can be seen as embodying a contemporary multiprogramming system in hardware. The microprocessors in a service center represent the system processes, while the program processors represent the active user processes. Unlike multiprogramming systems, however, there is no distinction between active and running processes.

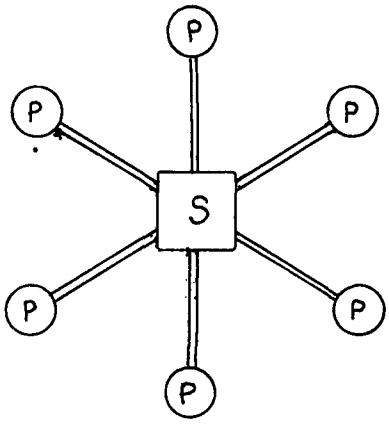


Figure 1.
S = service center
P = program processor

The service center performs four functions: 1) memory management, 2) process management, 3) file management (which includes all I/O) and 4) monitoring and protection. Each of these areas could be maintained by a separate microprocessor, called a service processor. Each of those microprocessors must have its own memory, both random access and control, and access to the status registers of all of the program processors it serves. Only the service processors have facilities for manipulating the system's resources. Obviously, the service processors may need to request service from each other. The four service processors and the program processors may be connected together by a single bus (see figure 2). Only one program processor may be receiving service at a time. If the technology can support it, the service center may be a multiple bus system. A program processor can request service only by interrupting a service processor. Its request is posted on the bus and an equitable arbitration system, such as one described by Chen [3], decides which of the program processors to attach to the service processor.

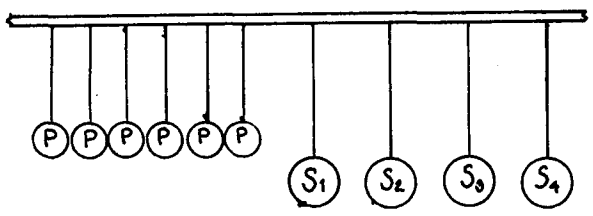


Figure 2. p = programmer
s_i = service processor i

To maintain efficiency and reduce interconnection complexity, it is desirable to limit the number of program processors attached to a service center. Since it is also desirable to have a very large number of program processors, a large-scale system must have a number of clusters. The simplest means for connecting an arbitrary number of clusters together seems to be the ring structure employed by the Distributed Computing System [6]. Since most processes are self-contained, the only need for communication between clusters is for multi-process tasks whose demand for processors cannot be met within the cluster that initiated the task. This sort of limited interaction is ideal for a ring system. The ring interface can be maintained by the process management processor in a service center. In addition to interprocess communication, there are other global system resources that must be shared by all clusters, namely the central memory and the input/output and file system. There is one bus connecting each cluster and the central memory. The memory manager controls all transactions. As in the UNIX system [18], for example, all input/output devices can be considered special cases of general files. Therefore, the file manager need only be concerned with one type of entity, a file. If the file system is maintained by a central file computer (like the PDP-10 serves the OCTOPUS system [2]) the operations of a file manager need be only formatting and posting requests to the file system computer.

Memory description. In order to avoid memory interference, it is necessary that a large memory be broken up into many modules. An obvious problem exists in trying to connect possibly hundreds of processors with an equally large number of memory modules. There are three principal means of interconnecting processors and memories [4]. Multi-port modules, in which each processor has a direct connection to every memory module, as in the IBM 360/370 series, is impractical because of the large number of interconnections and the inability to expand the number of modules beyond the number of ports. A crossbar system, in which each processor is connected to a switch which also connects all memories, as in C.mmp, has fewer connections, but expansion of the system is still difficult. A bus system, where processors and memories share (i.e., multiplex) one or more bus lines, seems the only practical means in terms of cost and expandability. To handle the large numbers of memory modules, a multi-bus system is proposed.

The switching logic required for this sort of system could be implemented with a standardized switch module. With each module would be associated an address, corresponding to the identifier of a memory module. As the main memory in the proposed system is block-oriented, the memory module address would be the high order bits of the addresses of the words within the module. The switch would have four ports the width of the principal data path (see figure 3). A memory request would come in through one port. If the high order bits matched the address of the memory module, the switch would connect the input bus with memory module bus and the data transfer could proceed. If the addresses did not match, then the switch would connect the incoming bus with one of the two remaining busses: if the requested address was less than the assigned address, the left port is chosen; otherwise, the right port is chosen. Clearly, if the busses of the memory modules were organized in a binary search tree manner (figure 4), then a processor could be connected to any memory module in time log N, where N is the number of memory modules. If the time of switching is much less than the expected time of memory transfer, then the switching time could be ignored. Memory modules may be added simply by filling up or extending the tree at the cost of one switch per memory module per processor. Likewise, each processor need be connected directly to only one switch to access the set of memory modules, and indirectly to at most 2**(N-1) switches, if there are 2**N memories.

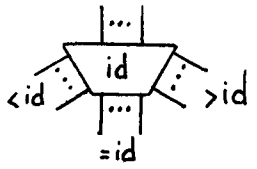
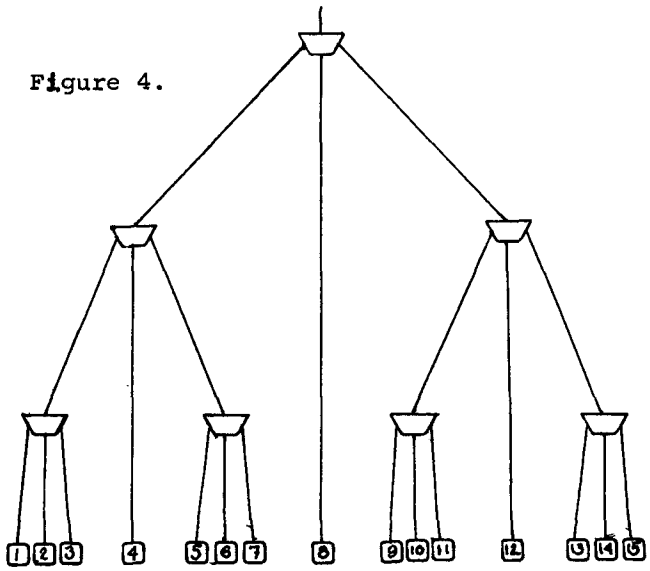


Figure 3.

Figure 4.



The cost of the processor-memory interconnection would be prohibitive if every one of the program processors had direct access to the memory modules. However, the program processors are constrained to access main memory through the memory management processor serving the relevant clusters. This reduces the number of interconnections by an order of magnitude. A single bus connects all memories of the program and service processors of a cluster and the topmost switch of the memory-switch tree, including a connection to the memory management processor. All transfers along this bus consist of single blocks.

Contention. The great problem with all multiprocessor systems is contention, and in a system with hundreds of processors a great deal of architectural ingenuity must be employed to circumvent it. There are basically two types of contention: memory interference and system contention. Memory interference results when two or more processes attempt to share a common region of code or data. The problem is program-dependent, and the responsibility for avoiding it may rest with the programmer. Techniques for parallel programming that recognize this problem are just beginning to be developed [12].

The situation is different with respect to system contention, which results when two or more processors request common system routines or tables. In fact, a principle motivation for the proposed system is to reduce system contention. By identifying the state of a process with a processor, a number of process control tables and task dispatcher routines are obviated. It is not immediately obvious that the memory management system can avoid the problems of contention for the global memory management system.

An extremely large central memory can simplify the problems of memory management and hence the problems of system contention in the memory management section of the operating system. Current systems like MULTICS, even with extended core storage, run out of primary memory space under heavy loads and must transfer pages to the file system. If the central memory is large enough so that it is unlikely to run out of room page traffic to the file system will be reduced considerably.

Although it is expected that there will be a large amount of sharing of data and procedures, there will always be a considerable amount of unshared code and data. For example, the runtime stack of an ALGOL program or the code for any undebugged procedure are unlikely to be shared. If all segments, where a segment is a collection of pages, as in MULTICS, are declared to be either shared or nonshared when they are created, then a subset of the memory modules may be partitioned among the clusters to that all nonshared segments associated with a program processor are assigned to memory modules allocated to the cluster containing the processor. No two clusters will have to access the same memory module within the subset (except in the case of message-passing between cooperating processes running on separate clusters), so the page tables for those modules need be maintained only by the associated memory management processor.

By the use of more LSI technology, it should be possible to construct associative memories which can be attached as peripherals to the memory management processors. With these memories, it is possible to condense page tables, segment tables and unallocated area tables into a single table. Every segment created by the system can be assigned a unique identifier, like a capability identifier. In this way, it is possible to reference any segment with a unique, invariant integer. An entry in the associative memory would contain one bit for empty or full, enough bits to represent the unique segment identifier, plus enough bits to represent the number of pages in a segment. The associative memory can only assign an entry, test for match on segment identifier, test for match on segment identifier and page number, or test for empty. Each memory management processor has an attached associative memory with entries for all the pages in the memory modules assigned to that processor. There is one table for all shared memory modules. The latter table is "read-mostly" since, as mentioned above, the main memory size is sufficiently large to contain the full memory region of all running processes except under very heavy load and hence segment traffic is expected to be relatively stable. Therefore, it should be possible to implement this table by attaching a copy to each memory management processor. The memory management processors may read their tables independently, while all tables are locked when a processor attempts to write into one of the locations. Contention for system tables is thus reduced so that it can only occur when a segment is entering or leaving the system, i.e., the only occasions for writing.

System Functions. It should be clear how the memory management system operates. A program processor generates a page fault whenever it attempts to reference a page that is external to its buffer memory, or, alternatively, when it attempts to write onto a shared page. The program processors post their request to the associated service processor. When service is granted, the service processor determines the page and segment number of the desired page, and whether a read or write is desired. Each program processor maintains a set of

registers, like base registers, which contain the unique segment identifiers of the segments referred to by the process running on the processor. A program processor address needs only enough bits to specify a segment register, a page within a segment and a displacement within a page. For buffer memory references, the segment register number can be used directly as an address to reduce the address space size in the program processor. The memory management processor concatenates the contents of the segment register and the displacements to form an integer which uniquely identifies a word within a segment. It then checks the associative memories to see if the page is present. If it is, the associative memory returns the memory module location and the transfer can be initiated. If it is not, then the missing page can be transferred into memory from the file system (where the page will be located at a fixed displacement from the start of the segment). The service processor does this by transferring the page into the first available entry the associative memory reports as empty. If there is no room, then some other page may be transferred to the file system.

The concept of placing process queues in hardware also simplifies interprocess communication. The busy form of waiting is justified, since a program processor is necessarily idle whenever its associated process is waiting for a critical region. An operation analogous to an iterated TEST AND SET instruction, instead of a more complex queue implementation of P and V, is sufficient for interprocess synchronization. Since program processors directly reference only buffer memories, the TEST AND SET must be an interrupt to the process management processor. Each time a program processor cycles on a shared variable, it joins the queue of program processors waiting for service. This is equivalent to a queue of processes waiting for a V operation, with the queue protocol built into the cluster's bus arbitration.

Model. As an aid in understanding the effect in a cluster of maintaining process state in a processor, as opposed to shared system tables, an M/M/M//N queueing model has been used. That is, a closed system with N processes, sufficient queueing (i.e., storage) capacity and M program processors. The service time distributions are Markovian, i.e., exponential. (See figure 5). An Assumption has been made that clusters behave relatively independently. The model is closed and has two nodes. The first node represents the central server of a cluster and is modeled by uniprocessor. The other node has M processors, where M is the number of program processors. Exponential service times are assumed throughout.

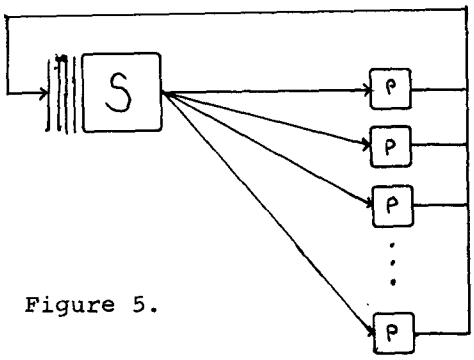


Figure 5.

S = service center
p = program processor

The service processor represents the "critical region" computation required for page faults and process state storage and initialization. It is assumed that the latter is, in time, about one half of the combined function and is modeled by the parameter α. More specifically, when there is an idle program processor the process state manipulation need not be carried out. The parameter rho represents the ratio of the arrival rate of service requests (λ) to the service time (μ). Assuming a 98 per cent hit ratio and an average program instruction time of 3 microseconds, a program processor would, on the average, generate a request for service every (3 x 100/2)=150 microseconds. Assuming a 25 microsecond time for a page transfer (projected for imminent CCD storage devices), a value for rho would be 150/25=6. Figures 6 and 7 illustrate the expected queue length at the service node for the number of processes, N, equal to 8 and 16, with three values of rho. The independent variable M is the number of program processors in the cluster. If k, the number of processes waiting or in computation at the M processors is taken as the state of the system and p_k is the probability of the system being in state k, then, from standard queueing theory [9], the exponential parameter for the service processors and the program processors are respectively:

$$\mu_k = \begin{cases} k\mu & 0 \le k < M \\ M\mu/\alpha & k \ge M \end{cases}$$

and

$$\lambda_k = \begin{cases} \lambda k & 0 \le k < N \\ 0 & \text{otherwise} \end{cases}$$

For $0 \le k \le M-1$,

$$p_k = p_0 \left(\frac{\lambda}{\mu}\right)^k \cdot \frac{1}{k}\ !$$

and for $M \leq k \leq N$,

$$p_k = p_0 \left(\frac{\lambda}{\mu}\right)^k \cdot \frac{1}{M!} \cdot \left(\frac{\alpha}{M}\right)^{k-M}$$

where p_0 may be computed in a straightforward manner.

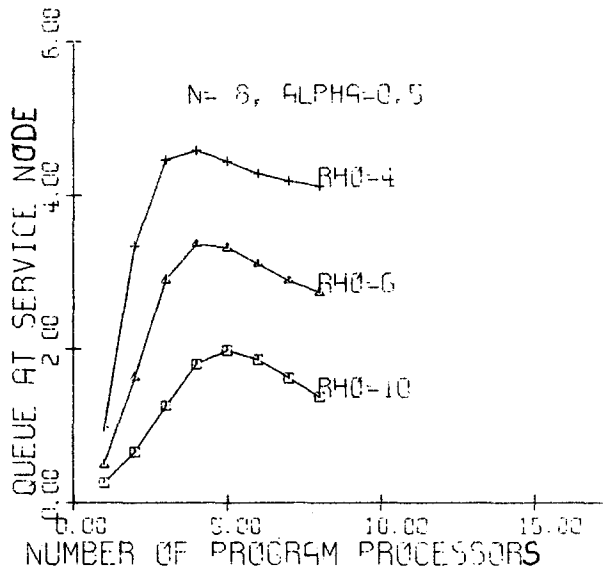


Figure 6.

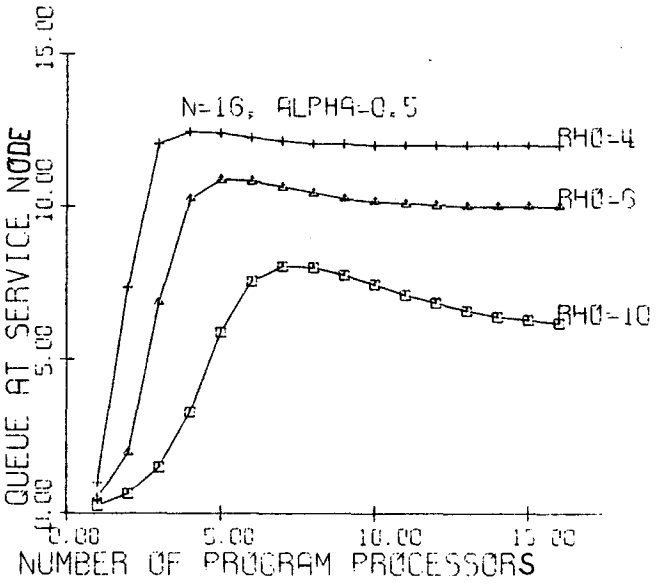


Figure 7.

It can be seen that the reduction of the need for the service processor to handle process state changes prevents the service node from overloading, as the queue length decreases as the number of program processors goes up (once it is likely that there is an idle program processor). In fact, because the curve is relatively flat, it is reasonable to add processes to a cluster beyond the number of program processors without losing the advantage of process-processor identification.

Conclusions. The incorporation of connection-limited microprocessors in a general purpose network is a difficult design problem. The approach suggested here is to identify system processes with microprocessors and interconnect them in a hierarchy constructed so that intercommunication requirements are small compared to independent processing time. This structure is examined in a preliminary way and needs a more elaborate quantitative analysis. Its feasibility depends critically on a number of service time distributions that can only be estimated at this time. Specifically, the size and performance of future buffer and bulk memories are very important.

References

1. Baum, A. and D. Senzig, Hardware considerations in a microcomputer multiprocessing system. IEEE COMPCON, Feb. 1975, 27-30.

2. Burk, J.M. and J.E. Schoonover, Computer system maintainability at the Lawrence Livermore Laboratory. Proc. AFIPS FJCC, part I, 1972, 263-272.

3. Chen, R. C-H., Bus communication systems, Ph.D. Thesis, Carnegie-Mellon University, January 1974.

4. Enslow, P., ed., Multiprocessors and Parallel Processing. Wiley, New York, 1974.

5. Fabry, R.S., Capability-based addressing, C. ACM, 17, 7, July 1974, 403-412.

6. Farber, D., Data ring oriented computer networks, in Rustin, Randall, ed. Computer Networks, Prentice-Hall, Englewood Cliffs, N.J., 1972.

7. Gelenbe, E., P. Tiberio and J.C.A. Boekhorst. Page size in deman paging systems. Acta Informatica 3, 1, 1973, 1-24.

8. Heart, F.E., S.M. Ornstein, W. R. Crowther, and W. Barker. A new minicomputer/multiprocessor for the ARPA network. Proceedings AFIPS NCC 42, 1973, 529-537.

9. Kleinrock, L. Queueing Systems, Vol. i: Theory, Wiley, New York, 1975.

10. Liptay, J.S. The cache, IBM Sys. J. 7, 1, 1968, 15-21.

11. Madnick, S. and T. Donovan. Operating Systems, McGraw-Hill, New York, 1974.

12. Mason, P.H. The design of programs for asynchronous multiprocessors, technical

report, Department of Computer Science, Carnegie-Mellon University, March 1975.

13. Newell, A. and G. Robertson. Some issues in programming multi-mini-processors, Technical Report, Department of Computer Science, Carnegie-Mellon University, January 1975.

14. Organick, E.I. The Multics System: An Examination of Its Structure, MIT Press, Cambridge, Mass., 1972.

15. Raphael, H.A. Distributed intelligence microcomputer design, IEEE COMPCON, February 1975, 21-26.

16. Ravindran, V.K. and T. Thomas. Characterization of multiple microprocessor networks, IEEE COMPCON, March 1973, 133-137.

17. Ritchie, D.M. and K. Thompson. The UNIX time-sharing system, C. ACM 17, 7, July 1974, 365-375.

18. Roberts, L. and B. Wessler. Computer network development to achieve resource sharing, Proceedings AFIPS SJCC 36, 1970, 543-549.

19. Wensley, J.H. The impact of electronic disks on system architecture, Computer 8, 2, February 1975, 44-48.

20. Withington, F.G. Beyond 1984: a technology forecast. Datamation 21, 1, January 1975, 54-73.

21. Wulf, W.A. and C.G. Bell. C.mmp--a multi-mini processor, Proceedings AFIPS FJCC 41, 1972, 765-778.

22. Wulf, W., E. Cohen, W. Corwind, A. Jones, R. Levin, C. Pierson and F. Pollack. HYDRA: the kernel of a multiprocessor operating system, C. ACM 17, 6, June 1975, 337-345.