Design of HM^{2}p—A Hierarchical Multimicroprocessor for General-Purpose Applications

KANG G. SHIN, MEMBER, IEEE, YANN-HANG LEE, STUDENT MEMBER, IEEE, AND J. SASIDHAR, MEMBER, IEEE

Abstract—**This paper presents a tree-structured multiprocessor called the hierarchical multimicroprocessor (HM^{2}p), each node of which is composed of a cluster of processor modules (PM's), common memory, DMA interface, switches, communication lines, and a data processor associated with it. The HM^{2}p consists of two different hierarchies, one for data processing and the other for data distribution, which provide clean, structured separation between processing components and user interface components.**

There are two levels of interprocessor communications in the HM^{2}p, an implementation of which is developed with the monitor concept. By examining the access pattern of shared hardware resources, we have modeled the performance of the HM^{2}p as a multichain closed queueing network. Using this queueing model, the performance falloffs due to shared hardware (e.g., processors, memory, and I/O devices) are also analyzed, and the optimum number of processors in each cluster is then determined.

Index Terms—**Hierarchical multiprocessor, monitor, performance falloff, processing/data distribution hierarchy, queueing model, synchronization.**

I. Introduction

CONTINUING advances in VLSI technology have made it attractive to interconnect many inexpensive microprocessors and memories to build a powerful, cost-effective computer, namely, a multimicroprocessor (M^{2}p). Potential benefits to be gained from an M^{2}p include improved cost performance resulting from the exploitation of parallelism in most algorithms and many inexpensive but powerful microprocessors and memories, enhanced fault tolerance by using many available processors in the M^{2}p as redundant spares, and a high degree of modularity which permits the M^{2}p to grow or shrink by addition or removal of modular components. However, the main question that still remains to be answered satisfactorily is whether the microprocessor can be utilized as a building block for large general-purpose computer systems, thereby achieving a higher performance/cost ratio as compared to traditional uniprocessors. Excellent surveys of existing multiprocessors can be found in [1] and [2]. The unsolved issues associated with multiprocessors are well discussed in [3].

Computational tasks in many applications such as engineering, socioeconomics, biology, medicine, etc., can be partitioned into smaller cooperating tasks (which we shall call processes). The cooperating processes are then clustered into a number of tightly coupled process groups that are hierarchically related to each other. Also, typical applications of computer systems—especially those related to information retrieval (such as information handling related to medical, law enforcement, social services, etc.)—impose a time-varying load that calls for a robust system. Tree-structured multiprocessors appear to have the potential to exploit the hierarchical nature of most computations [4]–[6] and are structurally flexible. This paper deals with one such tree-structured multiprocessor called the *hierarchical multimicroprocessor* (HM^{2}p). The HM^{2}p forms a tree, each node of which consists of a cluster of processor modules (PM's), shared memory, switches, DMA interface, and serial communication links. The HM^{2}p has two distinct hierarchies; one is for data processing (called the *P-hierarchy*) and the other is for data distribution (called the *D-hierarchy*). The necessity of including these two hierarchies becomes apparent when we review the Cm* architecture developed at Carnegie-Mellon University and the Hierarchical Multicomputer Organization at the State University of New York, Stony Brook.

The Cm* consists of a two-level hierarchy, its lower level is composed of computer modules (Cm's) which are grouped into a cluster via a time-shared bus (map bus), and its upper level comprises homogeneous clusters which are interconnected via intercluster buses [6]–[8]. The central idea for the Cm* operating system is the concept of task forces, i.e., large collections of executing processes that cooperate to accomplish a single purpose [8], [9]. The main drawback in Cm*, however, is the integration of the I/O units into the system. The I/O units are made dependent on individual computer modules, and this results in an unstructured operating system and gives rise to reliability and utilization problems. This, to some extent, has been solved by the Hierarchical Multicomputer Organization [10], [11] where the idea of separating the control and data moving functions has been proposed. In the HM^{2}p, this idea has been extended to include clean, structured data processing as well as I/O interface to the system.

Manuscript received January 7, 1982; revised June 17, 1982. This work was supported in part by the National Institute of Justice, U.S. Department of Justice, under Contract J-LEAA-014-78.

K. G. Shin and Y.-H. Lee were with the Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY. They are now with the Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109.

J. Sasidhar was with the Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, NY. He is now with Softech, Waltham, MA 02154.

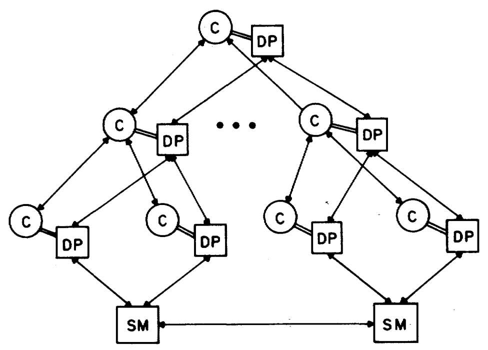


Fig. 1. System organization.

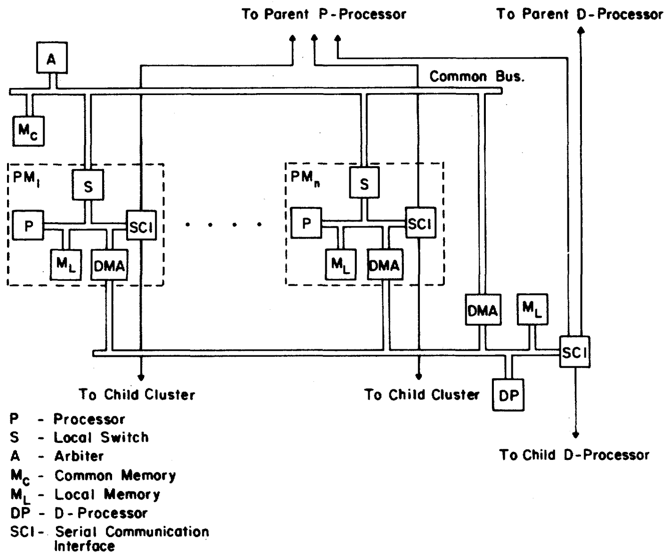


Fig. 2. Block diagram of a cluster with its associated D-processor in the HM^2p.

The clustering of processor modules in the HM^2p primarily is intended to provide hardware facility to execute a group of tightly coupled cooperating processes. This facility allows us to utilize the high degree of access locality [3] within the group. Saturation can, however, occur very quickly as the cluster expands. Indeed, it is a well-known fact that after a point, contentions for shared resources cause the performance of a cluster actually to decline with the addition of extra processors [12], [13]. It is therefore important to estimate the performance falloffs due to these contentions and then determine the optimum number of processor modules in each cluster. In this paper, we have studied this problem for the HM^2p using queueing network models.

For structural flexibility, the HM^2p has been designed to simplify the interconnection structure as far as possible. It uses only a few types of functional units as building blocks for the system. The aim of the design has been to create a general-purpose multiprocessor with no restriction on the types of algorithms which it can exploit.

This paper is organized as follows. Section II introduces the HM^2p architecture in some detail. Section III describes the structure of the kernels necessary to implement monitor primitives [14], [15] for synchronization purposes. Finally, Section IV analyzes the performance falloffs due to shared hardware resources, and the conclusion then follows in Section V.

II. Organization of HM^2p

The HM^2p is structured to accommodate the hierarchical nature of most computations, to exploit the parallelism existing within an algorithm, and to provide a systematic I/O interface to the system. The HM^2p consists of two distinct hierarchies: one for data processing (P-hierarchy) and the other for data distribution (D-hierarchy). The former is responsible for executing cooperating processes, and the latter is to distribute data necessary for the P-hierarchy. Both hierarchies consist of tree-structured clusters which are formed by a number of processors. To differentiate the processors in the P-hierarchy from those in the D-hierarchy, the former are referred to as the *P-processors* and the latter as the *D-processors.* The two hierarchies of the HM^2p are merged at the top by a root cluster (see Fig. 1 for an overall system organization).

A. Processing Hierarchy

The extent of exploitable parallelism with a multiprocessor depends on the overhead involved in communicating across process boundaries. The hardware interconnection scheme which has the lowest associated communication overhead is the shared memory method. The main drawback in using shared memory is that the communication overhead tends to increase rapidly with the number of processors in the multiprocessor. To alleviate this problem, the P-hierarchy is designed to include two levels of communication. At the first level, the communication time is kept to a minimum and is independent of the total number of processors in the HM^2p. Note that at this level, a restriction is imposed on the number of interconnected processors which are grouped into a cluster via a time-shared common bus and a shared common memory. At the second level, the communication speed is sacrificed for expansibility and hardware interconnection costs. At this level, the clusters are connected in a tree-structured fashion via serial links.

The significance of this approach becomes clearer when we consider process locality; interaction within a defined group of processes is generally frequent, and interaction between different groups is infrequent. If processes are assigned to processors such that the processes of the same task reside in a single cluster, then the communication overhead would correspond to that of a closely coupled system. This relates to the well-known task-assignment problem [16] in multiprocessors.

A cluster consists of processing modules (PM's) which have a sibling relationship to each other, and these PM's share a common memory by means of time-shared common bus (Fig. 2). Conflicts of access to the common bus are resolved by the bus arbiter. The handshaking required for gaining control of the bus is handled by the switch, which is a component of each PM. Each PM consists of a microprocessor, local memory, a switch, a DMA interface, and serial links to its child and parent PM's.

When the microprocessor issues a memory access, the switch in the concerned PM routes this access to its local memory or to the common memory. The common memory is used to store the system monitor parameters and the user-declared common segments that are shared by interacting processes (system synchronization will be discussed in Section III). As in the Cm*, processor-generated common memory addresses are translated by a window register [6] in the switch such that the user process is transparent to the physical allocation of shared segments. To perform a common memory access, the switch has to gain control of the common bus by handshaking with the arbiter. The switch has been given the capability of buffering a single data word which has to be read from or written into the common memory.

The arbiter has to be moderately complex since it must be able to grant control of the common bus at two separate levels (one for synchronization primitives and the other for common memory access), and there are certain rules to follow in order to preserve the integrity of the interprocess synchronization primitives (on this, more will be discussed later). The arbiter provides a round-robin service to requesting processors to ensure that all requests will be honored in due time. The switch in a PM has two request lines to the arbiter for requesting control of bus at the two levels, and correspondingly there are two grant lines to each switch. The switch includes a status to indicate whether its request is for synchronization or for access to the common memory. This status is explicitly set by the processor and is alterable only by the processor. Thus, the bus arbiter functions as either a coordinator or a global lock within a single cluster, depending on the nature of the currently honored request.

The DMA interface transfers a block of code/data to or from the local memory of the *D*-processor associated with the cluster and the local memories of the PM's. When the *D*-processor receives the block transfer order from the parent *P*-processor of the cluster, the *D*-processor then sets the address register and the word count register of the DMA interface appropriately and then initiates the transfer. Upon completing the transfer, the DMA interface notifies the *P*-processor which, in turn, notifies the parent *P*-processor of the cluster.

Besides the lateral interconnection paths, each *P*-processor has several serial links to its parent *P*-processor and its child cluster. With these serial links, clusters form a tree-structured hierarchy in which the number of branches at each node is equal to the number of *P*-processors in a single cluster. These links are used to transfer the control and status information between adjacent levels of the tree. They are also used to exchange messages between interacting processes that are located in different clusters. At both ends of the link, we need additional processing for buffering a message, generating interrupts, and setting up flags at the completion of a message transfer. A parent processor can interrupt its child processor through the serial link at two levels: one level is maskable and the other is nonmaskable. An interrupt at either of the two levels will cause the child to execute a message-receiving routine which is a part of the kernel software. In normal operation, a parent interrupts its child at the maskable level. This implies that if the child is inside the kernel, the interrupt will remain pending until the child exits from the kernel. But if the parent has reason to believe that a malfunction has occurred, it interrupts at the nonmaskable level. The child, on the other hand, can interrupt its parent through the serial control link only at the maskable level. This ensures that the parent can still function with a faulty child processor.

B. Data Distinction Hierarchy

For each cluster in the processing hierarchy, there is an associated *D*-processor which handles the transfer of code/data into or out of the cluster via the DMA channel. An additional link between the *D*-processor and the parent *P*-processor is provided to handle DMA commands and requests for process creation. The *D*-processors are interconnected to form the data distribution hierarchy as in Fig. 1. The secondary storage units as well as I/O devices are attached to the *D*-hierarchy. Since most of the processing is done at the bottom level of the *P*-hierarchy, most of the file transfers in the system will be handled by the associated leaf *D*-processors. Thus, we need high capacity data links between the secondary storage units and the bottom level *D*-processors of the *D*-hierarchy. To perform the file management functions of the system, the *D*-processors need to exchange short control messages among themselves. The *D*-processors are interconnected hierarchically by means of serial links, and since at times there will be file transfers on these links, a packet switching communication system has to be implemented.

All the user interfaces to the system are connected to the *D*-processors, and so it acts as the source of all tasks which need processing power from the processing hierarchy. New processes enter the *P*-hierarchy via the serial control links interconnecting the two hierarchies, and the computation results enter the *D*-hierarchy through the DMA channel. The *D*-processors act as command message interpreters in the same sense as the "shell" of the UNIX system [17] (called the *shell-like process*) and create processes which execute the command message in the processing hierarchy. Using the *D*-hierarchy and the links among *D*-processors, a new process can enter the *P*-hierarchy at any level. Although the cost of communication with I/O units may depend on their locations in the *D*-hierarchy, processor modules in the *P*-hierarchy are made logically independent so as to enhance the reliability of the overall system and to balance the work load in the *P*-hierarchy.

III. Synchronization and Communication

The objective of the operating system in the HM^{2}p is to allow maximum concurrency and transparency. Since the *P*-hierarchy includes the two-level structure—the closely coupled clusters and the tree network of clusters—we adopt two monitor systems associated with them. The processes are distributed in the system and communicate with each other via monitors and proper addressing mechanisms. Interprocess communication is based on the monitors and the kernels for the processes residing in a cluster and the monitor procedures in the tree network.

A. Kernels of the HM^2p Operating System

From a software point of view, monitors [8] are an ideal mechanism with which one can implement synchronization and interprocess communication. Monitors consist of shared data and procedures to operate on the shared data. A process can operate on the shared data only through the monitor procedure. The operations on the shared data are kept mutually exclusive. There are four primitives needed to support monitors, namely, entering a monitor, exiting a monitor, signaling a condition, and waiting for a condition. Since there are two levels of interprocess communication in the HM^2p, the kernels are designed to provide the execution environment at each level.

1) Kernels at the Cluster Level: For each P-processor, we define the processor kernel (called the *P-kernel*) which works as the supervisor of a processor to manage the process residing in that processor. The P-kernel has certain procedures to handle the execution of the user task. The task running in the P-processor of the cluster is in the user mode, and is trapped to the P-kernel when one of the following cases occurs: the invocation of a synchronization primitive, an addressing fault during a global access, an interrupt from the parent P-processor, or an exception. Through the P-kernel, the process can enter a monitor at the cluster level or the network level to perform synchronization and interprocess communication or request to declare a new space in the common memory. When the process enters the P-kernel, the interrupt is disabled to avoid a race condition.

The cluster kernel (called the *C-kernel*) handles the monitor of all processes residing in that cluster and is located in the common memory of that cluster. The mutual exclusion of the C-kernel is secured by the bus arbiter by means of the following steps.

a) If the process wants to enter the C-kernel, it sets a status bit in the switch. The switch then asserts the C-request line.

b) The bus arbiter asserts the C-grant line if the C-kernel is not in use. The switch then sets a flag indicating to the processor that it can now proceed to use the C-kernel.

c) Once the processor exits from the C-kernel, it resets the status bit in the switch which causes the switch to deassert the C-kernel request line.

The C-kernel provides mutual exclusion of the monitors by associating with each monitor a flag which records whether or not the monitor is busy. Thus, the C-kernel provides a means of having more than one monitor busy at the same time. The C-kernel maintains the queues for processes waiting to enter monitors and queues for each condition. There are four monitor primitives in the C-kernel as discussed above. When a process awaits a condition or a monitor, the P-processor executing this process becomes idle. When the process releases a monitor or signals a condition, it sends the identification of the waiting process to the parent P-processor. The parent P-processor then interrupts the processor which has that process in the wait queue. Thus, we have a "positive wakeup of a process" [18]. The C-kernel contains both the identification of the process and the physical processor in which it is residing, whereas the P-kernel keeps the full status of the process necessary to restart the process.

2) Kernel at the System Level: To implement the synchronization primitives at the system level, the monitor concept can still be used. This approach provides transparency and makes it easier for the system programmer to implement the system software. Since the clusters are arranged as a tree, we can always find a common ancestor cluster to interacting processes that are located in different clusters. The monitor procedure is allocated in the common ancestor cluster. The execution of a monitor procedure can be implemented via messages. The mutual exclusion is easily maintained since only one process is allowed to be in the monitor procedure of that ancestor cluster at one time.

B. Capability-Based Addressing

The objects of the system consist of resources and processes. The resources are passive elements which include files, I/O interfaces, processors, and memories. Resources are used for the execution of a process. In the HM^2p, each object is associated with its capability list which describes the environment of the execution and the relationship with other objects. A capability consists of a description of the concerned object, the associated operation, and the access right. Thus, the access of an object can be handled through this capability list in the execution space. The important advantages of the capability are to decentralize the overhead of address mapping [19] and object protection [20], and to treat the objects uniformly from the user's standpoint [17].

In the HM^2p, the addressing of the common memory in a cluster is translated by the switch. The parent P-processor partitions the common memory into several segments and provides the physical address to the concerned child P-processor. In order to address the I/O devices and the file system, the P-kernel maps the logical names to the physical locations, and also checks the access right on the basis of the capability of the object. At the destination, the condition of the object is checked to ensure the read/write consistency. Most passive objects reside in the data distribution hierarchy, and the D-processor has several I/O routines to provide the operations to access these objects.

When an I/O device is active, the shell-like process generates a local space (defined by the capability list), depending on the characteristics of the device and the user identity. All requests or commands are handled in the basic local space and its extension. When a process is invoked, a new space is generated including the capabilities associated with the process itself and the arguments. The legality of the process invocation is checked first in the basic space; then this invocation and the capability list are sent to the parent P-processor (the invoker may be the shell-like process in the D-processor or the process residing in the cluster). Either the loading of the process to an available processor or just migrating of the argument into an existing process would take place. If there is no available processor in the cluster, this invocation would be sent up to the grandparent P-processor, and a free processor would then be acquired from a neighboring cluster.

IV. Performance Analysis

The performance falloffs in a multiprocessor are generally introduced due to contention for use of shared hardware and software resources. Since the HM^{2}p consists of two levels (i.e., clusters and tree-structured entire system), and since most of the data processing and file management are carried out at the leaf levels of the data processing and distribution hierarchies, we first determine the performance of a single cluster, treating it as a stand-alone unit. Then we calculate the performance of the entire system by multiplying the number of leaf clusters by the performance of a cluster that is obtained from the single cluster analysis.

In the present work, the performance refers only to the throughput of the system, and is analyzed on the basis of the following two assumptions.

Assumption 1: Each cluster consists of a finite number of processors whose mechanisms of accessing the shared resources are statistically identical. All the clusters at the same level of the hierarchy thus have the same statistical property.

Assumption 2: No single processor in a cluster has the multiprogramming capability, while the cluster does have this capability. Thus, a new process would be assigned to a processor when the processor completes the previous assignment and becomes idle. The processor has to be suspended every time it awaits the availability of a shared resource.

A. Performance of a Single Cluster

A cluster in the HM^{2}p has a number of processors which are interconnected via the time-shared bus and communicate with each other via the common bus and common memory. This interconnection structure is very simple and flexible for expansion and reduction, but the system performance saturates quickly as the number of processors in a cluster increases mainly due to interprocessor communications. It is therefore important to analyze the performance of a single cluster which enables us to determine the optimum number of processors in a cluster and to calculate the performance falloffs due entirely to the shared hardware resources. In this paper, we do not consider the performance falloffs due to software precedence relationships which determine the final figure of performance.

The hardware resources shared by the *P*-processors of a cluster are the time-shared common bus-common memory pair, the parent *P*-processor, and the *D*-processor. Interference in sharing these resources results in a decrease in the performance of each processor. Taking this interference into consideration, we have developed a two-part queueing model for evaluating the performance of a single cluster. The first part expresses the performance falloffs due to common memory interference, and the second part models those due to the parent processor and *D*-processor of the cluster.

1) Common Memory Interference: Let the common memory access time (t_{ma}) be the time taken to read or write a single word into the common memory once the switch has mastership of the time-shared common bus in a cluster. Let the access request interval (t_{ar}) denote the time interval between two consecutive access requests to memory by a processor. The access requests can be either for code or data, and may also be made to either processor's local memory or the common memory of the cluster. As will be explained later, a processor may have to access the common memory at several consecutive times, for example, when it is granted to execute the *C*-kernel code. This fact eliminates the need of distinguishing the access request interval for the common memory from that for processor's local memory. Although there is a variation in the access request interval times, it is assumed for simplicity to be a constant as in [21].

Let us further denote the integer value $\lfloor t_{ar}/t_{ma} \rfloor$ by m where $\lfloor x \rfloor$ is the largest integer not exceeding x. Then one can observe that the greater the value of m, the less the interference due to the shared resource, and thus the greater is the performance of the processors in the cluster. With the current technologies of microprocessor and memory, the value of m is usually in the range 3–10 [22], and this can be used as a design parameter.

To analyze the interference in accessing the common memory, we should have an understanding of the nature of the stochastic process which describes the accesses to common memory by each processor. Reviewing the use of common memory we find that the common memory is used only for monitor procedures and their associated data and control mechanisms. When a processor starts executing a monitor procedure, all memory accesses will be to the common memory since both code and data reside in the common memory. Thus, successive accesses to common memory by the same processor cannot be modeled as independent random variables.

If a process executes any of the monitor primitives, it begins executing the code of the *C*-kernel, and then, depending upon the type of monitor primitives desired and the state of the desired monitor, one of the following actions takes place.

a) The processor starts to execute the monitor procedure.

b) It wakes up a process residing in another processor to execute the monitor procedure.

c) It waits for another process to signal it, and at that time it continues to execute the monitor procedure.

d) It does not execute the monitor procedure nor does it wake up another process to execute the monitor procedure.

In the first two cases, the monitor procedure is executed either by the same processor or by another immediately following the execution of the *C*-kernel. In the last two cases, the monitor procedures are not executed, and the next time the processor accesses the common memory, it would execute the *C*-kernel. Once a monitor procedure is being executed, the processor has to execute one of the monitor primitives to exit from the monitor. The above four cases can be condensed to the following case: the processor first executes the *C*-kernel, and then a monitor procedure, and then the *C*-kernel again. Finally, it returns to the local process, as shown in Fig. 3.

We can model the memory contention problem as a closed queueing network with appropriate service times and scheduling policies. The service time it provides can be measured in terms of the number of common memory accesses. The number of common memory accesses needed to execute a portion of a monitor procedure sandwiched between two consecutive

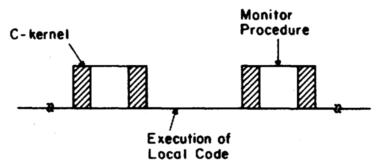


Fig. 3. Common memory reference pattern.

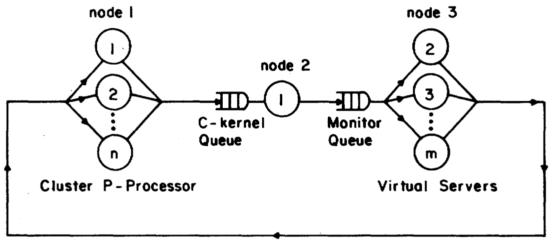
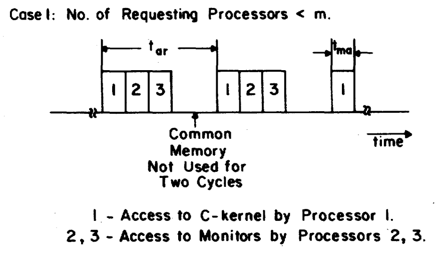


Fig. 4. Queueing model for common memory interference.


Case 1: No. of Requesting Processors < m.
t_{ar}
t_{ma}
Common Memory Not Used for Two Cycles
time
1 - Access to C-kernel by Processor 1.
2, 3 - Access to Monitors by Processors 2, 3.
Case 2: No. of Requesting Processors > m.
1 - Access to C-kernel by Processor 1.
2-6 - Access to Monitors by Processors 2 to 6.

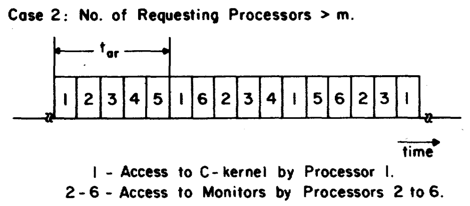


Fig. 5. Individual accesses to common memory.

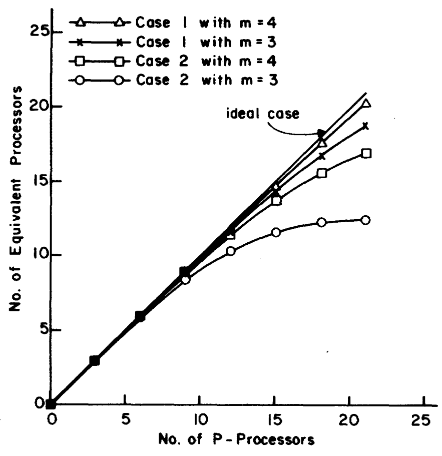


Fig. 6. The performance with common memory interference.

monitor primitives can be treated as a random variable with an exponential distribution. The number of local memory accesses between two monitor calls can also be treated as a random variable with an exponential distribution. The number of common memory accesses needed to execute the monitor primitive by means of the C-kernel is assumed to have an exponential distribution.

The queueing network consists of three nodes as shown in Fig. 4. The first node consists of n servers where n is the number of child P-processors in the cluster. The service time for these servers corresponds to the distribution of the number of local memory accesses between two consecutive monitor calls. Node 1 is of type-D [23] since the customers are delayed independently of other customers at this service station.

The common memory can be treated as m virtual parallel servers since effectively there can be m common memory accesses in time period t_{ar}. Also note that we cannot give more than one common memory access to a processor in a given time period t_{ar} (Fig. 5). Of the m virtual servers of the common memory, one server serves the C-kernel queue which is node 2 of the queueing network. The rest of the m-1 virtual servers serve the monitor queue and form node 3 of the queueing network.

In the actual system, however, the server serving the C-kernel queue would serve customers in the monitor queue if there is no customer in the C-kernel queue. Therefore, the performance characteristics obtained by this queueing network model gives a lower bound of the actual performance. The upper bound of the performance can be easily obtained by having an additional parallel server at node 3.

The scheduling policy used for nodes 2 and 3 of the queueing network is first-come-first-served (FCFS). In the actual system, the type of scheduling used to service the monitor queue is round robin. As we are only interested in the mean values of the waiting time and the mean queue lengths, we can assume an FCFS service mechanism. As long as the scheduling is independent of the service requirements of a customer, the mean values do not change [24].

The analytical solution of the queueing network was carried out by the recursive algorithm in [23]. The results shown in Fig. 6 correspond to mean service times indicated below (values normalized by the mean number of local memory accesses needed to execute a block of local code sandwiched between two consecutive monitor calls).

1) Mean percentage of common memory accesses needed for executing the C-kernel: case 1: 2.5 percent; case 2: 5 percent.

2) Mean percentage of common memory accesses needed for executing monitor code sandwiched between two monitor primitives: case 1: 10 percent; case 2: 20 percent.

These cases represent the access localities 89 and 80 percent respectively, which correspond to the experimental result obtained from Cm* [3]. The results give the lower and upper bounds of the performance of the cluster with common memory interference for $m=3$ and $m=4$.

2) *Parent P-Processor and D-Processor Interference:* To evaluate the interference in the parent P-processor and D-processor, the functions in both processors should be considered. When the process residing in a child P-processor requests the service from global resources (e.g., parent P-processor and data hierarchy), the process sends a message to the parent P-processor and waits for the reply. In the parent P-processor, the message handler accepts the request and puts the request

on a ready queue for service. Those services requested by the child *P*-processors are classified as one class with three types which are described as follows.

The first type is the synchronization request between processes residing in the same cluster. Since the processing time of this service is short and any delay will seriously affect the performance of the cluster, this request is given higher priority and relayed to the child *P*-processor immediately.

The second type of request is for execution of monitor procedures residing in the parent *P*-processor or communication with a processor in a different cluster. After recognizing this request, a new process is created and inserted into the ready queue of the parent *P*-processor. The processing time is longer, but the frequency of occurrence of this request is small.

The last type of request in this service class is the data transfer between the child *P*-processor and I/O devices or file system. The parent *P*-processor relays the request to the *D*-processor and then to the file system or I/O device. Upon the completion of data transfer, the *D*-processor informs the concerned parent *P*-processor, and then the blocked child *P*-processor resumes execution.

There are three other classes of services issued, respectively, by the grandparent *P*-processor, the I/O devices, and other *D*-processors in the data distribution hierarchy. The job assigned by the grandparent *P*-processor to the parent *P*-processor may be a user task that needs to be allocated to one of its offsprings, or a system supervisor task, or a communication message. In the *D*-processor, besides the data transfer, it provides the shell-like process for each I/O device and communication with other *D*-processors.

Considering interference at the shared resources, a multichain closed queueing network is developed to model the performance of an HM^{2}p cluster (Fig. 7). There are four chains with four different types of customers corresponding to the four service classes discussed above.

In the first chain, the requests issued by the child *P*-processor travel around the queueing network. Note that the model of common memory interference is also included in this network. Let p_{ijr} denote the branch probability from node i to node j following the service at node i in chain r. The process in the child *P*-processor (node 1) may request the use of either common memory with probability p_{121} or parent processor (node 4) with probability p_{141}. In the former case, the request goes to node 2 (*C*-kernel) and node 3 (monitor) and then returns to node 1. In the latter case, the parent *P*-processor services the request, which then follows one of the three paths to the next node. The path that directly returns to the child *P*-processor represents the synchronization service. The request is sometimes fed back to the parent *P*-processor in the second path in order to acquire further service, for example, the execution of the procedure or the communication with other clusters. The third path routes the request to the *D*-processor (node 5) or file system (node 6) because this request involves the data transfer via the DMA path to or from the I/O devices or file system. Corresponding to these paths, we define the three branch probabilities p_{411}, p_{441}, and p_{451}. Furthermore, after receiving the service from the *D*-processor, the request

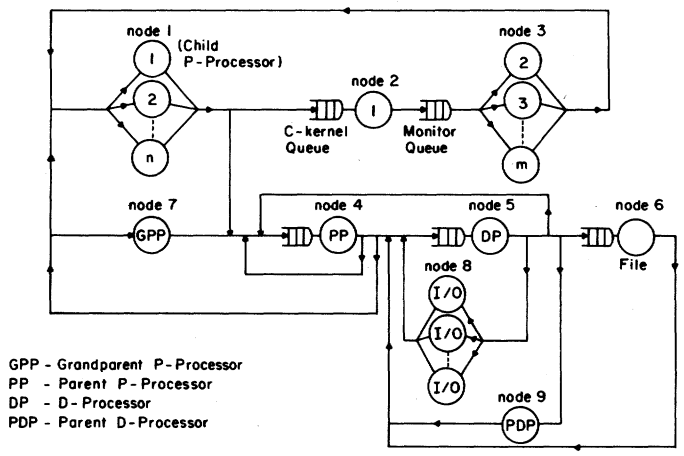


Fig. 7. Queueing model for performance of the HM^{2}p with interference in entire hardware resource.

may branch to the file system (node 6) or to the I/O devices for the data transfer between a child *P*-processor and an I/O device, (two branch probabilities p_{541}, p_{561} represent these two paths, respectively).

In the second chain, the parent *P*-processor accpets the request from the grandparent *P*-processor, and then returns the reply to the originator when the service is completed. The grandparent does not send another request until the previous service is completed. The third chain represents the shell-like process in the *D*-processor. The shell-like process would become active if an I/O device (node 8) issues a request. The fourth chain describes another function of the *D*-processor, namely, the communication in the data distribution hierarchy (in which the parent *D*-processor is represented as node 9). This implies that the interference due to the congestion in the *D*-hierarchy is also included in this model.

To solve this queueing problem, let us assume that node i ($i = 1, 2, \cdots, 9$) has an exponentially distributed service or request time with mean $1/\mu_{ir}$ for a customer in chain r ($r = 1, 2, 3, 4$). This service/request time may be a "think" time to make requests, an execution time, etc. For example, in node 1, the service/request time is the time interval between two external accesses; in node 4 it becomes the process execution time in the parent *P*-processor. From the branch probabilities p_{ijr} and the mean service/request rate μ_{ir}, the relative traffic density ρ_{ir} can be obtained where μ_{ir} is normalized such that $\mu_{11} = 1$. By assuming the service time in each node and the branch probabilities following each service, we can evaluate the total processing power for various cases. For instance, a small "think" time in the parent *D*-processor (node 9) represents a higher demand for file access from the *D*-hierarchy. A higher branch probability p_{141} and p_{441} will indicate heavier intercluster communications. Using the mean value algorithm in [23], the throughput of each chain is calculated (denoted as $1/\lambda_r$). The number of equivalent processors is then equal to $(\sum_{i=1}^{3} \rho_{i1})/\lambda_1$. The number of equivalent processors is calculated and then plotted in Fig. 8 for the following two cases:

$$\text{case 1: } \phi_1 = [1.0, 0.025, 0.1, 0.017, 0.027, 0.027]$$

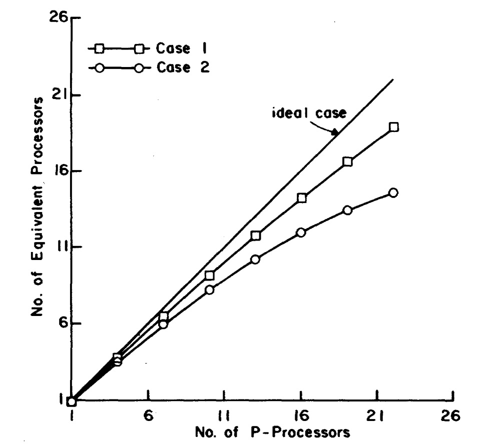


Fig. 8. The performance with interference in entire hardware resource.

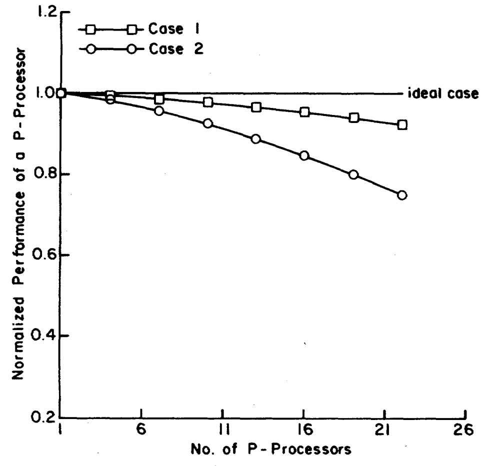


Fig. 9. Performance degradation versus the number of P-processors.

case 2: $\phi_1 = [1.0, 0.05, 0.2, 0.033, 0.053, 0.053]$.

For both cases

$$\phi_2 = [0.05, 0.5, 0.1, 5.0, 0.5, 0.6, 5.0]$$

and $m = 4$ where

$$\phi_1 = [\rho_{11}, \rho_{21}, \rho_{31}, \rho_{41}, \rho_{51}, \rho_{61}]$$

$$\phi_2 = [\rho_{42}, \rho_{72}, \rho_{53}, \rho_{83}, \rho_{54}, \rho_{64}, \rho_{94}].$$

The process localities used in cases 1 and 2 are the same as those in Fig. 6. Both cases are assumed to have the same interferences from the parent P-processor and D-processor, but the service times by shared resources are different (case 2 has service times twice those of case 1). The normalized performances of the child P-processor for both cases are plotted in Fig. 9.

B. The System Performance

From the above analysis, we have to arrive at the figure for the optimum number of processors in each cluster. Since m, the figure of merit of the common memory, can be varied within a reasonable range, one can design the HM^{2}p such that the parent P-processor and D-processor become the critical shared resources of a cluster.

Assuming that we desire at least 90 percent of the ideal performance (i.e., when there is no interference) and the value of m is equal to 4, we come up with the figure of 14 processors from the performance curves in Fig. 8 (case 1), which is the optimum number of P-processors for a single cluster. In this case, the net cluster performance is equivalent to that of 12.6 processors.

Most of the actual processing is assumed to take place in the leaf clusters, and the P-processors in the higher levels are thus busy synchronizing and performing other communication tasks. The total number of the leaf P-processors in the system is n^{k-1} where n is the number of P-processors in a cluster and k is the number of levels in the entire tree-structured system. Note that the total number of the P-processors in the system is $(n^k - 1)/(n - 1)$. If the number of child P-processors in a cluster is 13, then the equivalent number of the P-processors that are busy with the computation task only is 90 percent$*(13)^k$; this is almost 83 percent of the entire P-processors when k is large. On the other hand, the system provides $(n^{k-1} - 1)/(n - 1)$ P-processors to perform the communication task, task allocation, and system management. These constitute about 17 percent of the entire P-hierarchy.

V. Conclusion and Discussion

We have here presented the architecture of the HM^{2}p, the synchronization and communication primitives, and finally the performance of the system based on these primitives. The HM^{2}p is developed and analyzed based on a common feature of most computational tasks; namely, the tasks can be decomposed into cooperating processes which are then classified into closely coupled clusters, and these clusters hold a hierarchical relationship with each other. However, our simulation results for the performance analysis have indicated that the effectiveness of a multiprocessor heavily depends on the access locality. Thus, a key issue to any successful design of multiprocessors remains to be the development of an automatic method of decomposing a given task into interacting processes, and then assigning them to processor modules so that a high degree of access locality may be achieved.

As pointed out by one of the referees, it should be noted that the size of the local memory and the common memory will also affect both the access locality and the process allocation. (So does the size of cluster!) The access locality is therefore an important design parameter which is a function of the size of cluster, local memory, and common memory.

The processing power calculation for the HM^{2}p is useful not only for the leaf P-processors and their associated D-processors, but for the P-processors in the upper levels of the HM^{2}p. Since the P-processors in the upper levels are used for the communication and system supervision, the service times and the branch probabilities for these P-processors are different from those used for the leaf level. These processor modules in the upper levels may be regarded as a tree network used to exchange messages among cooperating processes. The communication overhead is implicitly included in the queueing model with the branch probability p_{441}, but needs further study for a more refined analysis. However, the branching factor for the upper levels may differ from that for the leaf level. The optimal values should be decided by examining the amount of communication (which is task-dependent). The higher level processors may become a bottleneck in the system if the amount of communication is heavy. It is also related to the number of the intercluster messages, which further depends on the structure of the operating system.

The effects of software precedence have to be introduced into our queueing models for determining the actual performance falloffs. This will be useful in determining the effects of both software and hardware constraints in the system. We know by intuition that the figure for the optimum number of processors in each cluster will increase when these effects are taken into account.

Acknowledgment

The authors are grateful to C. M. Krishna and the referees for the comments and criticisms on this paper.

References

[1] P. H. Enslow, "Multiprocessor organization—A survey," *Comput. Surveys*, vol. 9, pp. 103–129, Mar. 1977.

[2] M. Satyanarayanan, *Multiprocessors: A Comparative Study*. Englewood Cliffs, NJ: Prentice-Hall, 1980.

[3] S. H. Fuller, J. K. Ousterhout, L. Raskin, P. L. Rubinfeld, P. J. Sindhu, and R. J. Swan, "Multi-microprocessors: An overview and working example," *Proc. IEEE*, vol. 66, pp. 216–228, Feb. 1978.

[4] J. R. Goodman and C. H. Séquin, "Hypertree: A multiprocessor interconnection topology," *IEEE Trans. Comput.*, vol. C-30, pp. 923–933, Dec. 1981.

[5] S. A. Browning, "Computations on a tree of processors," in *Proc. VLSI Conf.*, California Inst. Technol., Pasadena, Jan. 1979.

[6] R. J. Swan, S. H. Fuller, and D. P. Siewiork, "Cm*—A modular multi-microprocessor," in *Proc. AFIPS Conf., Nat. Comput. Conf.*, vol. 46, 1977, pp. 637–644.

[7] R. J. Swan, A. Bechtolsheim, K. Lai and J. K. Ousterhout, "The implementation of Cm* multi-microprocessor," in *Proc. AFIPS Conf., Nat. Comput. Conf.*, vol. 46, 1977, pp. 645–655.

[8] A. K. Jones, R. J. Chansler, I. Durham, P. P. Feiler, and K. Schwars, "Software management of Cm*—A distributed multiprocessor," in *Proc. AFIPS Conf., Nat. Comput. Conf.*, vol. 46, 1977, pp. 657–663.

[9] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, "Medusa: An experiment in distributed operating system structure," in *Proc. 7th Annu. Symp. Operating Syst. Principles*, Nov. 1979, pp. 115–126.

[10] J. A. Harris and D. R. Smith, "Hierarchical multiprocessor organization," in *Proc. 4th Annu. Symp. Comput. Architecture*, Mar. 1977, pp. 41–48.

[11] R. B. Kieburtz, "A hierarchical multicomputer for problem solving by decomposition," in *Proc. 1st Int. Conf. Distributed Comput. Syst.*, Oct. 1979, pp. 63–71.

[12] W. W. Chu, D. Lee, and B. Iffla, "A distributed system for naval data communication networks," in *Proc. AFIPS Conf., Nat. Comput. Conf.*, vol. 47, 1978, pp. 783–793.

[13] C. J. Jenny, "Process partitioning in distributed systems," in *NTC Dig. Papers*, 1977, pp. 31:1–31:10.

[14] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, no. 10, pp. 549–557, 1974.

[15] R. C. Holt *et al.*, *Structured Concurrent Programming*. Reading, MA: Addison-Wesley, 1978.

[16] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, vol. 13, pp. 57–69, Nov. 1980.

[17] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 365–375, July 1974.

[18] H. K. Reghbati and V. C. Hamacher, "Hardware support for concurrent programming in loosely coupled multiprocessors," in *Proc. 5th Annu. Symp. Comput. Architecture*, Apr. 1978.

[19] R. S. Fabry, "Capability-based addressing," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 403–412, July 1974.

[20] E. Cohen and D. Jefferson, "Protection in the Hydra operating system," in *Proc. 5th Symp. Operating Syst. Principles*, Austin, TX, Nov. 1975.

[21] D. P. Bhandarker, "Analysis of memory interference in multiprocessors," *IEEE Trans. Comput.*, vol. C-24, pp. 897–908, Sept. 1975.

[22] B. Parasuraman, "High-performance microprocessor architectures," *Proc. IEEE*, vol. 64, pp. 851–859, June 1976.

[23] M. Reiser and S. S. Lavenberg, "Mean value analysis of closed multichain queueing networks," *J. Ass. Comput. Mach.*, vol. 27, pp. 313–332, Apr. 1980.

[24] L. Kleinrock, *Queueing System, Vol. 1 and 2*. New York: Wiley, 1975, 1976.

Kang G. Shin (S'75–M'78) was born in Choongbuk, Korea, on October 20, 1946. He received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, in 1976 and 1978, respectively.

From 1972 to 1974 he was on the Research Staff of the Korea Institute of Science and Technology, Seoul. From September 1978 to August 1982 he was an Assistant Professor in Computer Engineering, Department of Electrical, Computer, and Systems Engineering, Rensselaer Polytechnic Institute, Troy, NY. He has also taught short courses in the area of computer architecture for the Computer Science Series of the IBM Kingston Experienced Technical Education, Kingston, NY. Since September 1982 he has been a Faculty Member in the Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor. He is currently engaged in research and teaching in the areas of computer architecture, distributed and fault-tolerant computing, robotics and automation, and the application of control theory to biomedical systems.

Dr. Shin is a member of Sigma Xi and Phi Kappa Phi.

Yann-Hang Lee (S'81) received the B.S. degree in engineering science and the M.S. degree in electrical engineering from National Cheng Kung University, Taiwan, R.O.C., in 1973 and 1978, respectively.

Currently he is working towards the Ph.D. degree in electrical and computer engineering at the University of Michigan, Ann Arbor. His research interests include multiprocessor, performance evaluation, and fault-tolerant computing.

J. Sasidhar (S'80–M'81), photograph and biography not available at the time of publication.