(25)

C.mmp—A multi-mini-processor*

by WILLIAM A. WULF and C. G. BELL

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION AND MOTIVATION

In the Summer of 1971 a project was initiated at CMU to design the hardware and software for a multiprocessor computer system using minicomputer processors (i.e., PDP-11's). This paper briefly describes an overview (only) of the goals, design, and status of this hardware/software complex, and indicates some of the research problems raised and analytic problems solved in the course of its construction.

Earlier in 1971 a study was performed to examine the feasibility of a very large multiprocessor computer for artificial intelligence research. This work, reported in the proceedings paper by Bell and Freeman, had an influence on the hardware structure. In some sense, this work can be thought of as a feasibility study for larger multiprocessor systems. Thus, the reader might look at the Bell and Freeman paper for general overview and potential, while this paper has more specific details regarding implementation since it occurs later and is concerned with an active project. It is recommended that the two papers be read in sequence.

The following section contains requirements and background information. The next section describes the hardware structure. This section includes the analysis of important problem in the hardware design: interference due to multiple processors accessing a common memory. The operating system philosophy, and its structure is given together with a detailed analysis of one of the problems incurred in the design. One problem is determining the optimum number of "locks" which are in the scheduling primitives. The final section discusses a few programming problems which may arise because of the possibilities of parallel processing.

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-0107) and is monitored by the Air Force Office of Scientific Research.

REQUIREMENTS

The CMU multiprocessor project is designed to satisfy two requirements:

1. particular computation requirements of existing research projects; and
2. research interest in computer structures.

The design may be viewed as attempting to satsify the computational needs with a system that is conservative enough to ensure successful construction within a two year period while first satisfying this constraint, the system is to be a research vehicle for multiprocessor systems with the ability to support a wide range of investigations in computer design and systems programming.

The range of computer science research at CMU (i.e., artificial intelligence, system programming, and computer structures) constrains processing power, data rates, and memory requirements, etc.

(1) The artificial intelligence research at CMU concerned with speech and vision imposes two kinds of requirements. The first, common to speech and vision, is that special high data rate, real time interfaces are required to acquire data from the external environment. The second more stringent requirement, is real time processing for the speech-understanding system. The forms of parallel computation and intercommunication in multiprocessor is a matter for intensive investigation, but seems to be a fruitful approach to achieve the necessary processing capability.

(2) There is also a significant effort in research on operating systems and on understanding how software systems are to be constructed. Research in these areas has a strong empirical and experimental component, requiring the design and construction of many systems. The primary

IPR2023-00033
Apple EX1024 Page 1

requirement of these systems is isolation, so they can be used in a completely idiosyncratic way and be restructured in terms of software from the basic machine. These systems also require access by multiple users and varying amounts of secondary memory.

(3) There is also research interest in using Register Transfer Modules (RTM's) developed here and at Digital Equipment Corporation (Bell, Grason, et al., 1972) and in production as the PDP-16 are designed to assist in the fabrication of hardware/software systems. A dedicated facility is needed for the design and testing of experimental system constructed of these modules.

TIMELINESS OF MULTIPROCESSOR

We believe that to assemble a multiprocessor system today requires research on multiprocessors. Multiprocessor systems (other than dual processor structures) have not become current art. Possibly reasons for this state of affairs are:

1. The absolutely high cost of processors and primary memories. A complex multiprocessor system was simply beyond the computational realm of all but a few extraordinary users, independent of the advantage.
2. The relatively high cost of processors in the total system. An additional processor did not improve the performance/cost ratio.
3. The unreliability and performance degradation of operating system software,—providing a still more complex system structure—would be futile.
4. The inability of technology to permit construction of the central switches required for such structures due to low component density and high cost.
5. The loss of performance in multiprocessors due to memory access conflicts and switching delays.
6. The unknown problems of dividing tasks into subtasks to be executed in parallel.
7. The problems of constructing programs for execution in a parallel environment. The possibility of parallel execution demands mechanisms for controlling that parallelism and for handling increased programming complexity.

In summary, the expense was prohibitive, even for discovering what advantages of organization might overcome any inherent decrements of performance. However, we appear to have now entered a technological domain when many of the difficulties listed above no longer hold so strongly:

1′. Providing we limit ourselves to multiprocessors of minicomputers, the total system cost of processors and primary memories are now within the price range of a research and user facility.
2′. The processor is a smaller part of the total system cost.
3′. Software reliability is now somewhat improved, primarily because a large number of operating systems have been constructed.
4′. Current medium and large scale integrated circuit technology enables the construction of switches that do not have the large losses of the older distributed decentralized switches (i.e., busses).
5′. Memory conflict is not high for the right balance of processors, memories and switching system.
6′. There has been work on the problem of task parallelism, centered around the ILLIAC IV and the CDC STAR. Other work on modular programming [Krutar, 1971; Wulf, 1971] suggests how subtasks can be executed in a pipeline.
7′. Mechanisms for controlling parallel execution, *fork-join* (Conway, 1963), P and V (Dijkstra, 1968), have been extensively discussed in the literature. Methodologies for constructing large complex programs are emerging (Dijkstra, 1969, Parnas, 1971).

In short, the price of experimentation appears reasonable, given that there are requirements that appear to be satisfied in a sufficiently direct and obvious way by a proposed multiprocessor structure. Moreover, there is a reasonable research base for the use of such structures.

RESEARCH AREAS

The above state does not settle many issues about multiprocessors, nor make its development routine. The main areas of research are:

1. The multiprocessor hardware design which we call the PMS structure (see Bell and Newell, 1971). Few multiprocessors have been built, thus each one represents an important point in design space.
2. The processor-memory interconnection (i.e., the switch design) especially with respect to reliability.

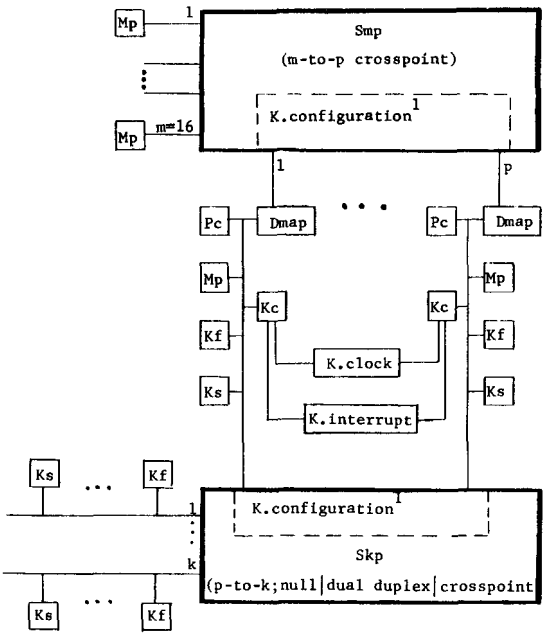
IPR2023-00033
Apple EX1024 Page 2

3. The configuration of computations on the multiprocessor. There are many processing structures and little is known about when they are appropriate and how to exploit them, especially when not treated in the abstract but in the context of an actual processing system:
   Parallel processing: a task is broken into a number of subtasks and assigned to separate processors.
   Pipeline processing: various independent stages of the task are executed in parallel (e.g., as in a co-routine structure).
   Network processing: the computers operate quasi-independently with intercommunication (with various data rates and delay times).
   Functional specialization: the processors have either special capabilities or access to special devices; the tasks must be shunted to processors as in a job shop.
   Multiprogramming: a task is only executed by a single processor at a given time.
   Independent processing: a configurational separation is achieved for varying amounts of time, such that interaction is not possible and thus doesn't have to be processed.
4. The decomposition of tasks for appropriate computation. Detailed analysis and restructuring of the algorithm appear to be required. The speech-understanding system is one major example which will be studied. It is interesting both from the multiprocessor and the speech recognition viewpoints.
5. The operating system design and performance. The basic operating system design must be conservative, since it will run as a computation facility, however it has substantial research interest.
6. The measurement and analysis of performance of the total system.
7. The achievement of reliable computation by organizational schemes at higher levels, such as redundant computation.

THE HARDWARE STRUCTURE

This section will briefly describe the hardware design without explicitly relating each part to the design constraints. The configuration is a conventional multiprocessor system. The structure is given in Figure 1.

There are two switches, Smp and Skp, each of which provide intercommunication among two sets of components. Smp allows each processor to communicate with all primary memories (in this case core). Skp



where: Pc/central processor; Mp/primary memory; T/terminals;
Ks/slow device control (e.g., for Teletype);
Kf/fast device control (e.g., for disk);
Kc/control for clock, timer, interprocessor communication

[1]Both switches have static configuration control by manual and program control

Figure 1—Proposed CMU multiminiprocessor computer/C.mmp

allows each processor (Pc), to communicate with the various controllers (K), which in turn manage the secondary memories (Ms), and I/O devices transducers (T). These switches are under both processor and manual control.

Each processor system is actually a complete computer with its own local primary memory and controllers for secondary memories and devices. Each processor has a Data operations component, Dmap, for translating addresses at the processor into physical memory addresses. The local memory serves both to reduce the bandwidth requirements to the central memory and to allow completely independent operation and off-line maintenance. Some of the specific components shown in Figure 1 are:

K.clock: A central clock, K.clock, allows precise time to be measured. A central time base is broadcast to all processors for local interval timing.

K.interrupt: Any processor is allowed to generate an interrupt to any subset of the Pc configuration at any of several priority levels. Any pro-

IPR2023-00033
Apple EX1024 Page 3

cessor may also cause any subset of the configuration to be stopped and/or restarted. The ability of a processor to interrupt, stop, or restart another is under both program and manual control. Thus, the console loading function is carried out via this mechanism.

Smp: This switch handles information transfers between primary memory processors and I/O devices. The switch has ports (i.e., connections) for m busses for primary memories and p busses for processors. Up to min(m,p) simultaneous conversations possible via the cross-point arrangement.

Smp can be set under programmed control or via manual switches on an override basis to provide different configurations. The control of Smp can be by any of the processors, but one processor is assigned the control.

Mp: The shared primary memory, Mp, consists of (up to) 16 modules, each of (up to) 65k, 16 bit, words. The initial memories being used have the following relevant parameters: core technology; each module is 8-way interleaved; access time is 250 nanoseconds; and cycle time is 650 nanoseconds. An analysis of the performance of these memories within the C.map configuration is given in more detail below.

Skp: Skp allows one or more of k Unibusses (the common bus for memory and i/o on an isolated PDP-11 system) which have several slow, Ks (e.g., teletypes, card readers), or fast controllers, Kf, (e.g., disk, magnetic tape), to be connected to one of p central processors. The k Unibusses for the controllers are connected to the p processor Unibusses on a relatively long term basis (e.g., fraction of a second to hours). The main reasons for only allowing a long term, but switchable, connection between the k Unibusses and the processor is to avoid the problem of having to decide dynamically which of the p processors manage a particular control. Like Smp, Skp may be controlled either by program or manually.

Pc: The processing elements, Pc, are slightly modified versions of the DEC PDP-11. (Any of the PDP-11 models may be intermixed.)

Dmap: The Dmap is a Data operations component which takes the addresses generated in the processor and converts them to addresses to use on the Memory and Unibusses emanating from the Dmap. There are four sets of eight registers in Dmap, enabling each of eight 4,096 word blocks to be relocated in the large physical memory. The size of the physical Mp is 2^{20} words (2^{21} bytes). Two bits in the processor, together with the address type are used to specify which of the four sets of mapping registers is to be used.

Dmap

The structure of the address map, is described below and in Figure 2 together with its implications for two kinds of programs: the user and the monitor programs. For the user program, the conventional PDP-11 addressing structure is retained—except that a program does not have access to the "i/o page," and hence the full 16-bit address space refers to the shared primary memory.

A PDP-11 program generates a 16-bit address, even though the Unibus has 18-bit addressing capability. In this scheme the additional two address bits are obtained from two unused program status (PS) register bits. (Note, this register is inaccessible to user pro-

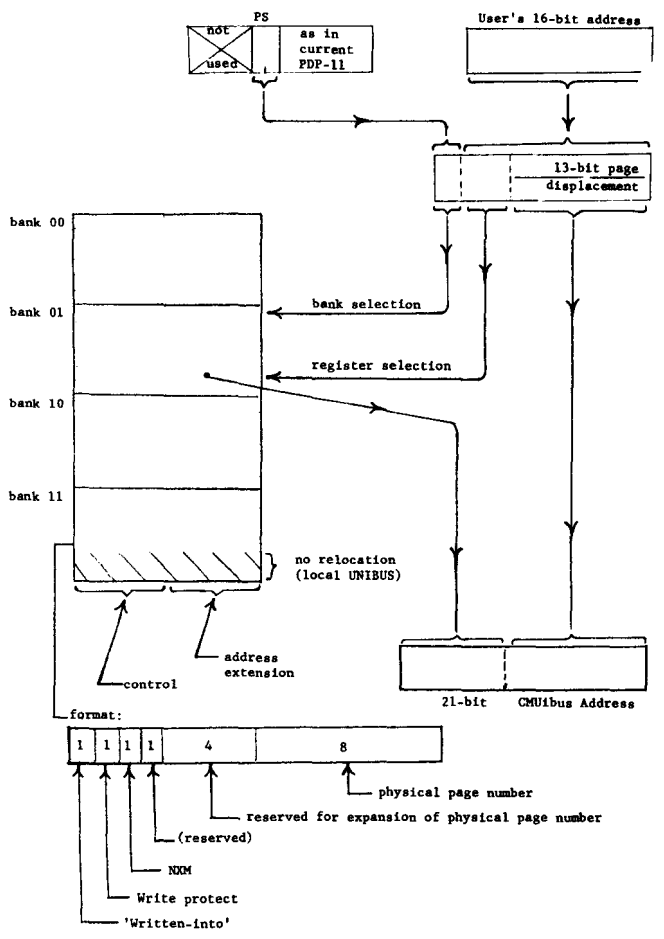


Figure 2—Format of data in the relocation registers

grams.) These are two additional bits, provides four addressing modes:

00-mode, 01-mode, 10-mode	These addresses are always mapped, and always refer to the shared, large, primary memory.
11-mode	All but 8 kw (kilo words) of this address space is mapped as above. The 8 kw of this space which is not mapped refers to the private Unibus of each processor; 4 kw of this space is for private (local) memory and 4 kw is used to access i/o devices attached to the processor.

For mapped references, the mapping consists of using the most significant five bits of the 18-bit address to select one of 30 relocation registers, and replacing these by the contents of the 8 low order bits of that register yielding an overall 21-bit address. Alternatively, consider that two bits of the PS select one of four banks of relocation registers and the leftmost three bits of the users (16-bit) address select one of the eight registers in this bank (six in bank three). A program may (by appropriate monitor calls) alter the contents of the relocation registers within that bank and thus alter its "instantaneous virtual memory"—that is, the set of directly addressable pages. The format of each of the 30 relocation registers is as also shown in Figure 2 where:

1. The 'written-into' bit is set (to 1) by the hardware whenever a write operation is performed on the specified page.
2. The 'write protect' bit, when set, will cause a trap on (before) an attempted write operation into the specified page.
3. The NXM, 'non-existent memory', when set, will cause a trap on any attempted access to the specified page. Note: this is not adequate for, nor intended for, 'page fault' interruption.
4. The 8-bit 'physical page number' is the actual relocation value.

THE MEMORY INTERFERENCE PROBLEM

One of the most crucial problems in the design of this multiprocessor is that of the conflict of processor requests for access to the shared memories.

Strecker (1970) gives closed form solutions for the interference in terms of a defined quantity, the UER (unit execution rate). The UER is, effectively, the rate memory references and, for the PDP-11, is approximately twice the actual instruction execution rate. (Although a single instruction may make from one to five memory references, about two is the average.) Neglecting i/o transfers*, assuming access requests to memories at random, and using the following mean parameters:

t_p	the time between the completion of one memory request and the next request
t_a, t_c	the access time and cycle time for the memories to be used
$t_w = t_c - t_a$	the rewrite time of the memory

Strecker gives the following relations:

$$t_p = t_w: \quad \text{UER} = (m/t_c)\,(1 - (1 - 1/m)^p)$$

$$t_p < t_w: \quad \text{UER} = \frac{m}{t} \times \frac{1 - (1 - 1/m)^p}{1 - (1 - 1/m)^p}$$

$$t_p > t_w: \quad \text{UER} = (m/t_c)(1 - (1 - P_m/m)^p)$$

$$\text{where } P_m + (m/p)\left(\frac{t_p - t_r}{t_c}\right)(1 - (1 - P_m/m^p)) - 1 = 0$$

Various speed processors, various types of memories, and various switch delays, t_d, can be studied by means of these formulas. Switch delays effects are calculated by adding to t_a and t_c, i.e., $t_a' = t_d + t_a$; and $t_c' = t_d + t_c$. For example, the following cases are given in the attached graphs. The graphs show UER $\times 10^6$ as a function of p for various parameters of the memories. The two values of t_d shown correspond to the estimated switch delay in two cable-length cases: 10′ and 20′. The t_c, t_a values correspond to six memory systems which were considered. The value of t_p is that for the PDP-11 model 20.

Given data of the form in Figures 3 and 4 it is possible to obtain the cost effectiveness of various processor-memory configurations. An example of this information for a particular memory configuration (16 memories, $t_c = 400$) and three different processors (roughly corresponding to three models of the PDP-11 family) is plotted in Figure 5. Note that a small configuration of five Pc.1's has a performance of 4.5×10^6 accesses/second (UER). The cost of such a system is approximately $375K, yielding a cost-effectiveness of 12. Replacing these five processors with the same number of Pc.3's yields a UER of 15×10^6 for about $625K, or a cost-effectiveness of about 24. Following this strategy provides a very cost-effective system once a reasonably large number of processors are used.

* A simple argument indicates that i/o traffic is relatively insignificant, and so has not been considered in these figures. For example, transferring with four drums or 15 fixed head disks at full rate is comparable to one Pc.

IPR2023-00033
Apple EX1024 Page 5

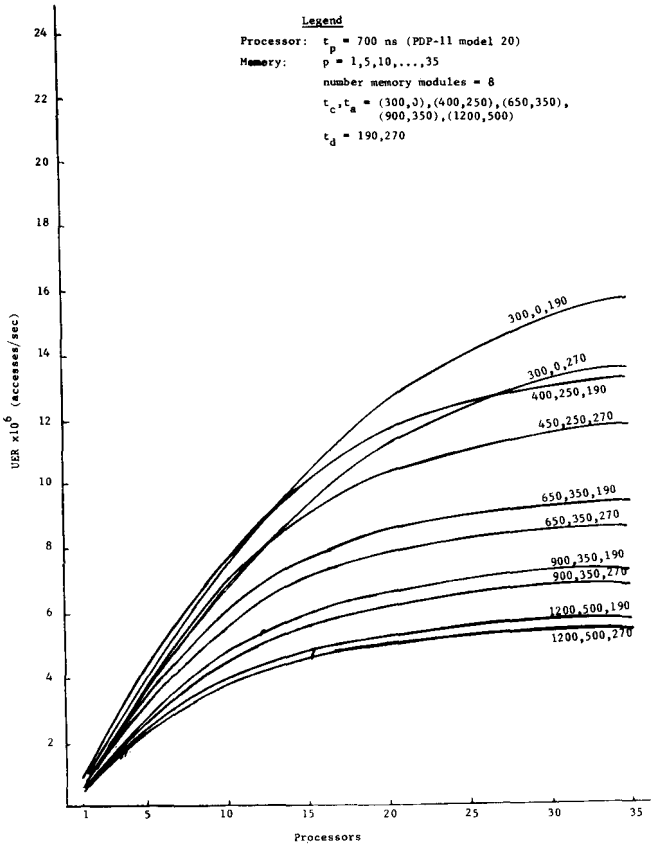


Figure 3—Performance for various memory-processor configurations

In fact, in the range 15–30 processors the cost-effectiveness is relatively constant while the absolute performance nearly doubles.

Unfortunately these studies of memory interference assume a random distribution of memory references—an assumption may be invalid when true parallel processing is performed (notably if shared programs are executed, as in the operating system). Several approaches to predicting and preventing these conflicts are being studied:

Software page-placements

Better-than-random reference patterns may be achieved by having the operating system page-placement algorithms attempt to localize process' pages within a single memory module. No results on this approach have been obtained to date.

Switch, Smp, measurement

Schemes for dynamically measuring the Mp-Pc reference pattern are being considered. The most accurate method under consideration is to associate a small memory with each crosspoint intersection. This can be constructed efficiently by having a memory array for each of the m rows, since control is on a row (per memory) basis. When each request for a particular row is acknowledged, a 1 is added to the register corresponding to the procesor which gets the request. These data could then serve as input to algorithms of the type described under (1). Such a scheme has the drawback of adding hardware (cost) to the switch, and possibly lowering reliability. Since the performance measures given earlier are quite good, even for large numbers of processors, this approach does not seem justified at this time.

A cache

Since performance for all but shared programs may approximate the random references assumption of Strecker's analysis, special provision for these references might be provided. The addition of a cache memory between Dmap and Smp allows programs to migrate

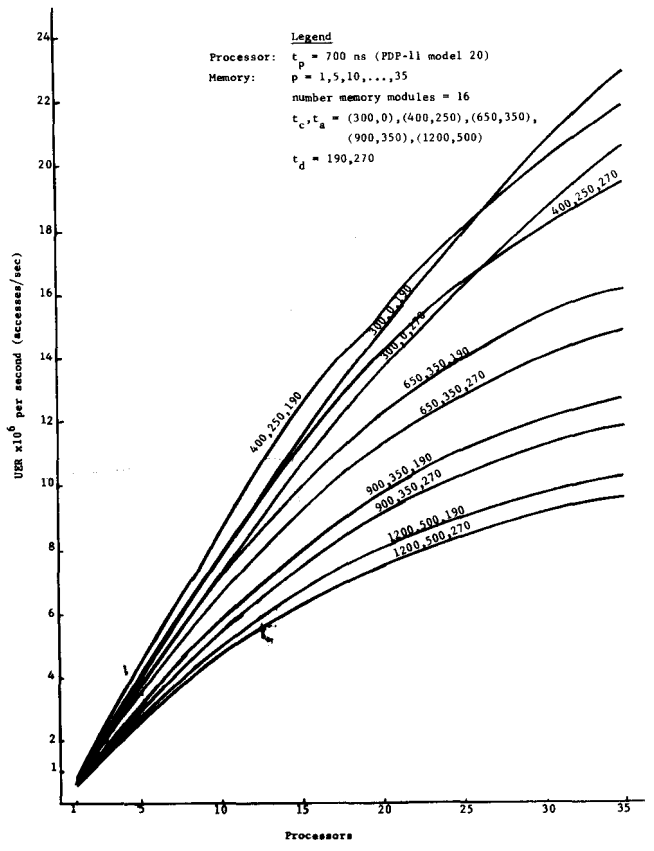


Figure 4—Performance for various memory-processor configurations

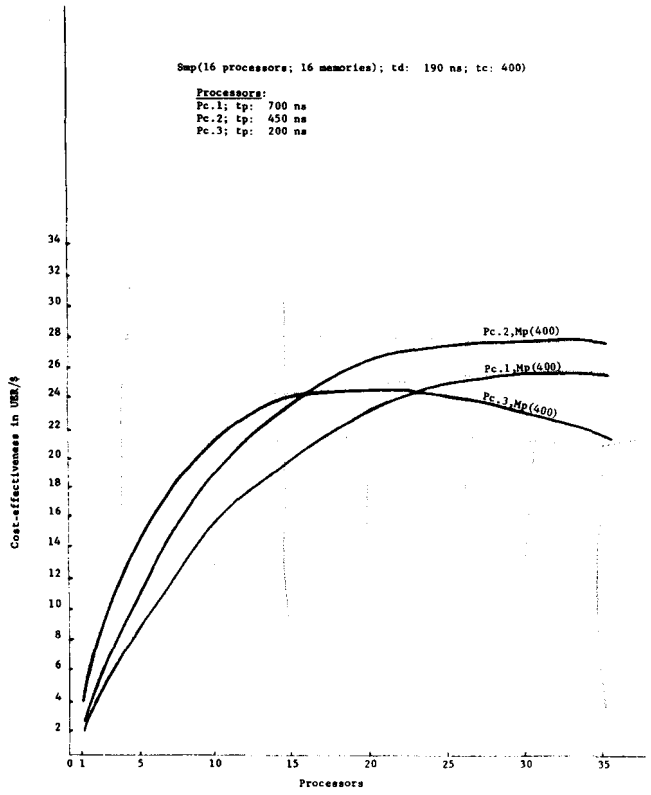


Figure 5—Cost effectiveness (UER/$)

into the cache thereby diminishing the number of requests for a single memory. This also provides faster access since the Smp is avoided.

By introducing such a cache, however, a potential problem is created regarding the validity of data since it might be possible to have sixteen different values of a single variable at a given instant of time. A scheme for avoiding this is to allow only information from "read only" pages (especially instructions) to appear in the cache. (In particular, the bit marked 'reserved' in Figure 2 is used to signal that data from the page may be placed into the cache.) Traces of PDP-11 programs executions indicate that a small cache (256–512 words) will capture 70–90 percent of the eligible references and 40–50 percent of all references. McCredie (1972) has studied the effect of such a cache on overall system performance both analytically and by simulation. The results of these studies indicate an improvement of 10–40 percent in overall system performance.

THE OPERATING SYSTEM

Although the technology of operating systems has made significant progress in the past decade, there are virtually no extant examples of systems constructed specifically for multiprocessor environments. In particular, no systems have been built to support the variety of process relations (parallel, pipeline, etc.) envisioned for C.mmp. Moreover, there is a relative lack of experience in organizing computations for parallel execution. These facts have driven the operating system design to the following, hopefully conservative, position:

> The operating system will consist of a "kernel" and a "standard Extension." The kernel will provide a set of mechanisms (tools) for building an operating system, but no policies (e.g., no scheduler, no file structure, no. . .). The standard extension will implement an (easily modified or replaced) set of "conventional" operating system facilities (e.g., a scheduler, file system, . . .). The kernel will support the (simultaneous) execution of an (almost) arbitrary number of extensions.

Under this strategy the variety of computational structures is not *a priori* limited by the structure of the underlying system. There are also potential hazards in the kernel approach. One of them is the possibility that extension in some (important) desired direction is not possible because of irrevocable decision made too early (though this problem is hardly unique to the kernel approach). Another hazard is that intolerable overhead might accrue by enforced multiple 'layering' of extensions. Both analysis and simulated use indicated that neither of these problems exist for the proposed design.

The remainder of this section is devoted primarily to a description of the kernel (called HYDRA).

In considering what set of mechanisms (tools) should be provided by an operating system kernel two commonly held views of the essential nature of an operating system are relevant:

—An operating system creates a "virtual machine" to support (user) programs by providing resources and operations not present in the underlying hardware (e.g., "files," file "read" and "write" operations, etc.).
—An operating system is a resource (virtual and physical) manager and allocator.

Note the emphasis in both views on resources; their creation, management, and operations on them. From these views we infer than an appropriate set of tools for building an operating system must provide for:

—the creation of new virtual resources;
—the 'representation' of a new resource in terms of existing ones;

—the creation of operations on resources and/or their representation; and
—protection (against illegal operations on a resource), both
  (a) uniformly over a class of resources; and
  (b) with regard to specific instances of a resource.

This list serves as the design goals for HYDRA against which the design is evaluated. .

Since the resources are central to the design, we define a suitable abstraction of these called an *object*; objects are the basic entity of interest in HYDRA. An object has a *name*, a *type*, and usually some other (type dependent) information associated with it. The name of every object is unique and is called its *global name*. There is a supply of unique global names to last over the system's total life. Thus, it is not possible for two (or more) objects to have the same global name.

The set of extant objects is partitioned into equivalence classed by their types. There is also an unlimited supply of object types—new types may be created at will. The initial system includes a particular object whose name is TYPE. New types are created by creating an object whose type is TYPE; thus a class of objects of a particular type are "represented" by an object of type TYPE. Suppose, for example, one wished to create a new kind of virtual resource. This would be done by creating an object (assume its name is X) of type TYPE. The object X now serves as a representative for all particular instances of resources of this new variety; in particular, objects of type X may now be created to represent the instances of the new resource.

Operations are performed on objects by procedures.* A procedure is an object of type PROCTYPE. The 'right' to invoke a procedure on each particular object is limited by both the type of object and the user's access to it (see below).

During execution of a procedure there exists a *local name space, lns*, associated with it. The *lns* is an object which provides a mapping between local object names (integers) accessible to the procedure and the actual global names for objects. Each *lns* entry may also restrict the access rights (procedures that can be invoked to perform operations on or with the object) to a subset of those defined for that type of object. Thus the *lns* provides both mapping and protection functions.

* Here we wish to invoke the reader's intuitive notion of a 'procedure' and its properties, e.g., a body of code, local storage, a parameter mechanism, etc.

The only primitive operations in the system which are *not* provided by procedures are CALL and RETURN, whose functions are, respectively, to permit entry to, and exit from, procedures. CALL also provides parameter checking and establishes the *lns* for the called procedure.

To recap: The primitive notions in HYDRA are those of an *object*, a *global name*, and a *type*. Some specific types are TYPE, PROCTYPE, and LNSTYPE. Procedure objects may be invoked by a CALL and are exited by a RETURN. Protection is provided by: (1) restricting access to objects to those named in the current *lns*, (2) restricting the operations (procedures) which may be applied to an object to those associated with that type of object, and (3) further restricting the set of operations which may be applied to any object named in an *lns* to a subset of those in (2).

Figure 6, gives a concrete example of this mechanism. Suppose that a new type of object, a "bibliography file," is created. Three specific operations are permitted on these objects: updating, printing, and erasing. Therefore three procedure objects UPDATE, PRINT, and ERASE are created to perform these

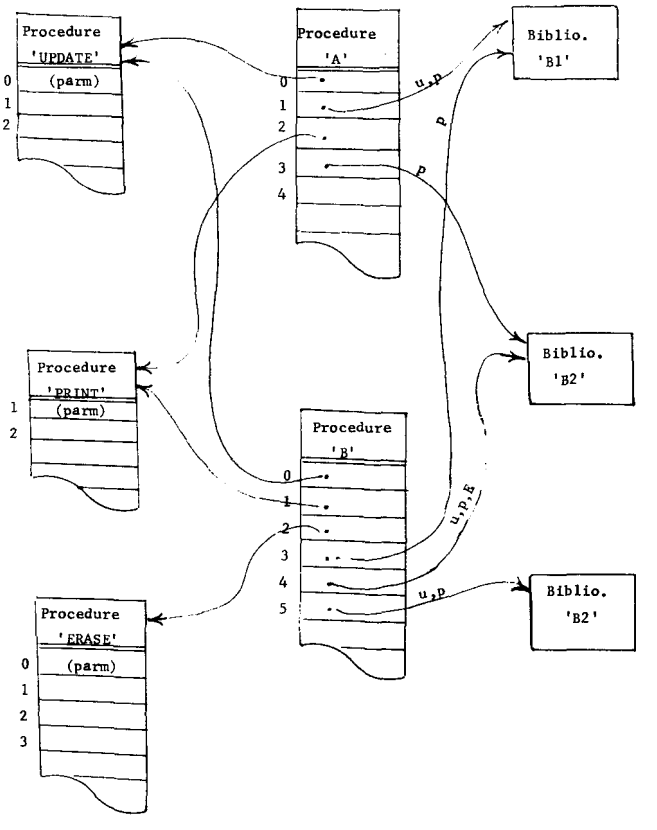


Figure 6—Example of LNS mapping and protection

operations; no other operations are permitted on this type of object. The situation in Figure 6 might exist at some instant. It shows (in the center) two procedures, A and B, and their associated lns's—directed arcs indicate the mapping function of the *lns* and the letters along an arc indicate permitted accesses. Here, local name '1' of procedure A references a particular bibliography object, B1; UPDATE and PRINT access by A are permitted. The following information can be observed from the diagram:*

—A.0 → UPDATE
—B.0 → UPDATE
—A.2 → PRINT
(and so on; note that A cannot name the ERASE procedure nor bibliography object B2)
—A may: update and print B1; only print B2
—B may: only print B1; update, print, or erase B2; update and print B3.

THE RELIABILITY PROBLEM

The existence in the physical system of multiple, redundant resources suggests the possibility of highly reliable operation—at least in the sense of continuing to provide (degraded) service when some fraction of the hardware is down. An explicit goal in the HYDRA design is to provide commensurate reliability in the software. Reliability may have two components:

(1) Correctness: The major reason for unreliability in current software is that it is incorrect. However,
—the proposed design for the kernel is small enough that a "constructive programming" approach can be used effectively (Dijkstra)
—the design suggests natural modular decomposition along the lines suggested by Parnas (Parnas 1972)
—the coding is being done in a "systems implementation language" (Bliss/11) (Wulf, et al., 1970, 1971)
—the protection mechanism itself absolutely guarantees that an erroneous or malicious program cannot destroy information to which it does not have legal access.
Therefore the correctness of the kernel must be proven and its construction is proceeding in a highly stylized form design to facilitate this.

(2) Malfunction: Even if the software is correct it is possible for the system to be unreliable, for example, as the result of misexecution of correct code by (perhaps intermittently) failing hardware. This problem is compounded by both the multiprocessor character of the system and the kernel design.

Although a great deal of research has been done on hardware reliability, (for example in connection with computers for extended space missions and electronic telephone switching systems), little has been done on software reliability. Undoubtedly this situation has resulted from the fact correctness (or lack of it) rather than malfunction has been the primary cause of unreliable software.

Possibly some of the ideas from the work on hardware reliability can be carried over to software; a few of these are discussed below. It should be remembered that there is a cost/effectiveness trade-off in each of these—an increasing degree of reliability may be achieved only at an increased cost. A very high degree of reliability appears expensive and probably unnecessary in any case.

Redundancy

One of the common forms of fault detection is to replicate a critical component and, at appropriate points, to verify that the components agree. This might appear in several forms in software:

—Critical computations might be performed by two distinct methods within a single processor and their results compared
—The same code for a critical computation might be performed by two distinct processors and their results compared
—Multiple copies of critical data might be stored on distinct devices and their contents compared.

Consistency

A less demanding (and expensive) form of fault detection is to merely check the reasonableness of a computation or data item value. A simple example is for all lists to be stored in "circular, doubly-linked" form since this permits a check that the predecessor and successor of an item correctly point to the item. Another example of the same kind is for critical items to carry a "self-identification" which is checked before any updates to the item are made.

* The notation X.n will be used to refer to the *n*th local name in procedure *X*; "→" is to be read "maps onto" or "is a reference to."

IPR2023-00033
Apple EX1024 Page 9

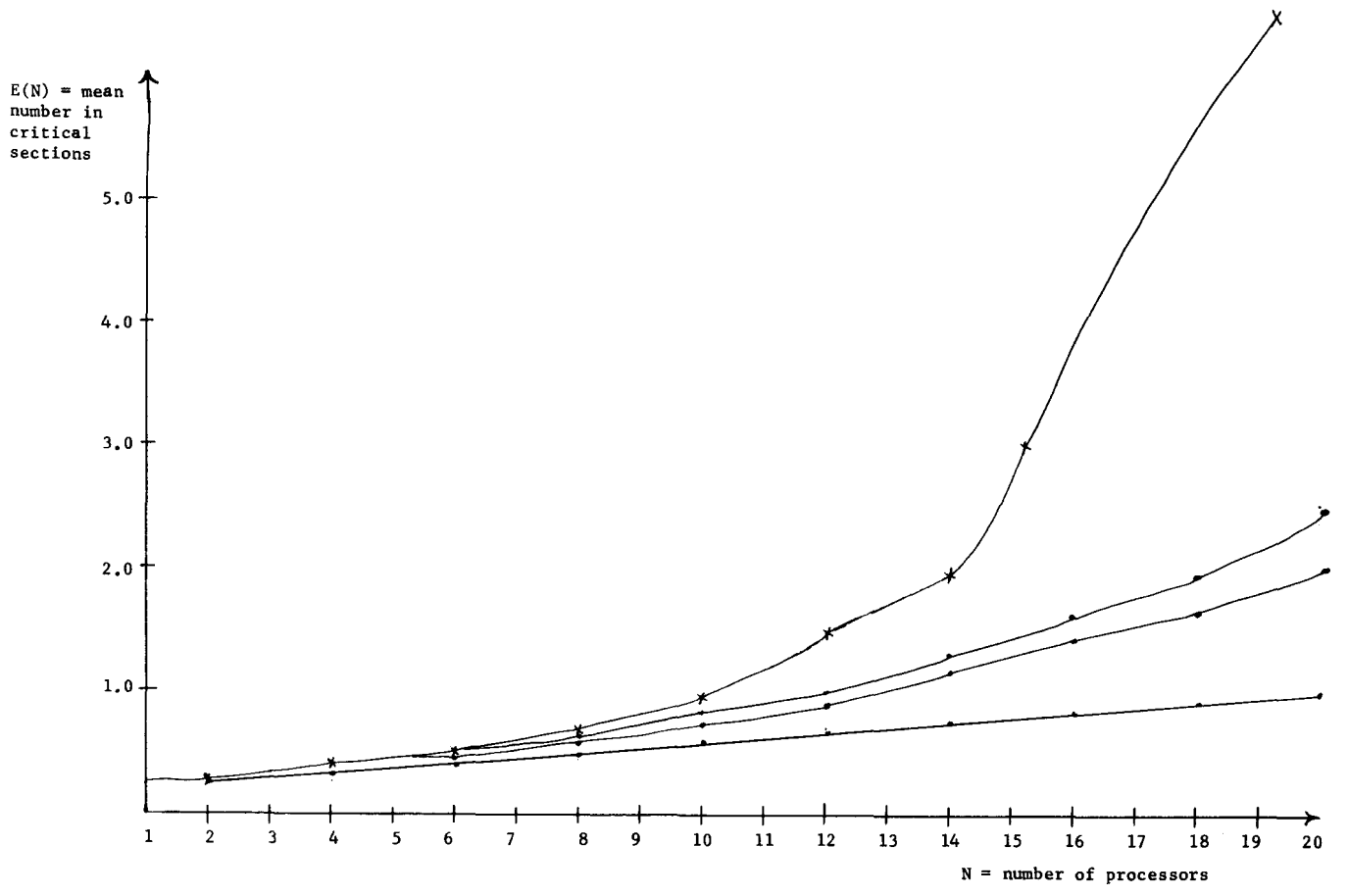


Figure 7—Mean response time for scheduling

Diagnostics

An even less demanding scheme is to attempt to ascertain whether the hardware is functioning properly before faults occur in critical places. This might be done on the fly just before a critical computation is performed, at fixed intervals, or simply whenever the processor is not occupied with other tasks.

THE LOCK PROBLEM

An interesting problem in the design of a multiprocessor operating system is scheduling and coordinating the many, individual processors. In HYDRA the information necessary to make these decisions is represented in a shared data base and the program(s) which make the decision may be executed on any of the processors—and possibly on several processors simultaneously. While one processor is accessing or updating this shared information all other processors must be prevented from accessing and/or changing it. The act of protecting a data item is called "locking" and that portion of a program which accesses a locked item is called a "critical section."

A basic design problem in such a scheduler is to determine the number of critical sections, S, that will maximize system performance. At one extreme a single lock could be used for the entire data base; at the other extreme each item could have a separate lock. Since a finite time, L, is required to perform the locking operations, the overhead due to this operation is minimized for $S = 1$. However, if $S = 1$, the possibility of several processors performing scheduling operations simultaneously is precluded. Even though the performance for each individual processor is degraded, total system performance may be improved by choosing $S > 1$.

A report by McCredie (McCredie, 1972) discusses two analytic models which have been used to study this problem; here we shall merely indicate the results. Figure 7 illustrates the relationship predicted by one of McCredie's models between the mean response time to a scheduling request, the number of critical sections, and the number of processors.

Mean response time increases with the number of processors. For S constant, the increase in mean response time is approximately linear, with respect to N, until the system becomes congested. As N increases beyond this point, the slope grows with increasing N.

The addition of one more critical section significantly improves mean response, for higher values of N, in both models. The additional locking overhead, L, associated with each critical section degrades performance slightly for small values of N. At these low values of N, the rate of requests is so low that the extra locking overhead is not compensated for by the potential parallel utilization of critical sections.

The most interesting characteristic of these models is the large performance improvement achieved by the creation of a small number of additional critical sections. The slight response time degradation for low arrival rates indicates that an efficient design would be the implementation of a few (S=2, 3 or 4) critical sections. This choice would create an effective safety valve. Whenever the load would increase, parallel access to the data would occur and the shared scheduling information would not become a bottleneck. The overhead at low arrival rates is about 5 percent percent and the improvement at higher request rates is approximately 50 percent.

Given the dramatic performance ratios predicted by these modes, the HYDRA scheduler was designed so that S lies in the range 2–7 (the exact value of S depends upon the path through the scheduler).

PROGRAMMING ISSUES

Thus far both highly general and highly specific aspects of the hardware and operating system design of C.mmp have been described. These alone, however, do not provide a complete computing environment in which productive research can be performed. An environment of files, editors, compilers, loaders, debugging aids, etc., must be available. To some extent existing PDP-11 software can and will be used to supply these facilities. However, the special problems and potentials of a multiprocessor preclude this from being a totally appropriate set of facilities.

The potential of true parallel processing obviously requires the introduction of language and system facilities for creating and synchronizing sub-tasks. Various proposals for these mechanisms have existed for some time, such as fork-join, "P" and "V", and they are not especially difficult to add to most existing languages, given the right basic hardware. Parallelism has a more profound effect on the programming environment, however, than the perturbations due to a few language constructs. The primary impact of parallelism is in the increase in complexity of a system due to the possible interactions between its components. The need is not merely for constructs to invoke and control parallel programs, but for conceptual tools dealing with the complexity of programs that can be fabricated with these constructs.

In its role as a substrate for a number of rearch projects, C.mmp has spawned a project to investigate the conceptual tools necessary to deal with complex programs. The premise of this research is that the approach to building large complex programs, and especially those involving parallelism, is essentially methodological in nature: the primitives, i.e., language features, from which a program is built are not nearly as important as the *way* in which it is built. Two particular methodologies—"top-down design" or "structured programming" (Dijkstra, 1969) and "modular decomposition" (Parnas, 1971) have been studied by others and form starting points for this research.

While the solution to building large systems may be methodological, not linguistic, in nature, one can conceive of a programming environment, including a language, whose structure facilitates and encourages the use of such a methodology. Thus the context of the research has been to define such a system as a vehicle for making the methodology explicit. Although they are clearly not independent, the language and system issues can be divided for discussion.

Language issues

Most language development has concerned itself with "convenience"—providing mechanisms through which a programmer may more conveniently express computation. Language design has largely abdicated responsibility for the programs which are synthesized from the mechanisms it provides. Recently, however, language designers have realized that a particular construct, the general *goto*, can be (mis)used to easily synthesize "faulty" programs and a body of literature has developed around the theoretical and practical implications of its removing from programming languages (Wulf, 1971a).

At the present stage of this research it is easier to identify constructs which, in their full generality, can be (mis) used to create faulty programs than to identify forms for the essential features of these constructs which cannot be easily misused. Other constructs are:

Algol-like scope rules

The intent of scope rules in a language is to provide protection. Algol-like scope rules fail to do this in two ways. First, and most obviously, these rules do not distinguish kinds of access; for example, "read-only" access is not distinguished from "read-write" access. Second, there is no natural way to prevent access to a variable at block levels "inside" the one at which it is declared.

Encoding

A common programming practice is to encode information, such as age, address, and place of birth, in the available data types of a language, e.g., integers. This is necessary, but leads to programs which are difficult to modify and debug if the manipulation of these encodings is distributed throughout a large program.

Fixed representations

Most programming languages fix both syntactic and run-time representations; they enforce distinctions between macros and procedures, data and program, etc., and they provide irrevocable representations of data structures, calling sequences, and storage allocation. Fixed representations force programmers to make decisions which might better be deferred and, occasionally, to circumvent the fixed representation (e.g., with in-line code).

SYSTEMS ISSUES

Programming should be viewed as a process, not a timeless act. A language alone is inadequate to support this process. Instead, a total system that supports all aspects of the process is sought. Specifically, some attributes of this system must be:

(a) To retain the constructive path in final and intermediate versions of a program and to make this path serve as a guide to the design, construction, and understanding of the program. For example, the source (possibly in its several representations) corresponding to object code should be recoverable for debugging purposes; this must be true independent of the binding time for that code.

(b) To support execution of incomplete programs. A consequence of some of the linguistic issues discussed above is that decisions (i.e., code to implement them) will be deferred as long as possible. This must not preclude compilation and testing of portions of a program which do not depend on earlier decisions.

(c) To integrate a file system into the constructive process. In particular the file maintenance of the system must have the responsibility of maintaining the structure of programs, the correspondence between different representations of the same program, keeping track of cross-references between files, distributing information from modules to compilers, etc.

SUMMARY

We have attempted to outline the need and goals for the multiprocessor computer system being constructed at CMU. The hardware and software structure were presented in overview form, together with detailed analysis of various critical parts. We believe that such a system is one which will become important in the future, simply because of the capabilities it provides and the way in which it utilizes technology.

ACKNOWLEDGMENT

A significant fraction of the faculty, students, and staff, of the Department of Computer Science at CMU are either directly or indirectly involved or have made significant contributions to this project. It is as difficult to give them all full credit as it would be incorrect to assume the authors are the source of all ideas or work reflected in this paper. Those most directly involved have been:

Professors Allen Newell and Raj Reddy who provided most of the initial motivation and served in continuing review; Bill Broadley, the manager of the engineering lab, who has designed and is constructing the specialized hardware; Chuck Pierson, who has responsibility for coordinating the project; Ellis Cohen, Roy Levin, Bill Corwin, and Fred Pollack who are programming the

operating system; Anita Jones, whose insights lead to much of the operating system philosophy; Professor Jack McCredie who has developed analytic models for the memory interference and lock problems, and Professor Mary Shaw who is developing the programming system described in the last section.

REFERENCES

1 C G BELL R CADY H McFARLAND B DELAGI J O'LAUGHLIN R NOONAN W WULF
A new architecture for minicomputers—The DEC PDP-11
SJCC 1970 pp 657-675

2 C G BELL P FREEMAN et al
C.ai: A computing environment for AI Research—Overview, PMS, and operating system considerations
Department of Computer Science Carnegie-Mellon University May 1971

3 C G BELL J GRASON S MEGA R VAN NAARDEN P WILLIAMS
The design, description and use of the DEC register transfer modules (RTM)
IEEE Transaction on Computers May 1972

4 C G BELL A N HABERMANN J McCREDIE R RUTLEDGE W WULF
Computer networks
Computer Science Research Review Carnegie-Mellon University 1969

5 C G BELL A NEWELL
Computer structures
McGraw-Hill Book Company 1971a

6 C G BELL A NEWELL
Possibilities for computer structures, 1971
FJCC 1971b

7 M CONWAY
A multiprocessor system design
Proceedings of the IFIP Congress Yugoslavia 1971a

8 *DEC PDP-11 documents*
Programmer Reference Manual and Unibus Interface Manual

9 E DIJKSTRA
Cooperating sequential processes
In Programming Languages F Genuys (ed) Academic Press 1968

10 E DIJKSTRA
Structured programming
Software Engineering October 1969 Rome

11 R KRUTAR
Personal Communication 1971

12 D McCRACKEN G ROBERTSON
C.ai (P.L)—a L* processor for C.ai*
Department of Computer Science Carnegie-Mellon University Pittsburgh 1971

13 J McCREDIE
Analytic models as aids in multiprocessor design
Department of Computer Science Carnegie-Mellon University Pittsburgh 1972

14 D L PARNAS
On the criteria to be used in decomposing systems into modules
Department of Computer Science Report Carnegie-Mellon University Pittsburgh 1971

15 W D STRECKER
An analysis of the instruction execution rate in certain computing structures
PhD Dissertation Carnegie-Mellon University ARPA Report 1971

16 W WULF
Programming without the goto
Proceedings of the IFIP Congress Yugoslavia 1971a

17 W WULF et al
A software laboratory: Preliminary report
Department of Computer Science Carnegie-Mellon University Pittsburgh 1971

18 W WULF et al
Bliss reference manual
Department of Computer Science Report Carnegie-Mellon University Pittsburgh 1971

19 W WULF D RUSSELL A N HABERMANN
Bliss: A language for systems programming
Communications of the ACM December 1971