Chapter 22

The C.mmp/Hydra Project: An Architectural Overview

Henry H. Mashburn

Summary This article describes the C.mmp/Hydra project at Carnegie-Mellon University. Included are detailed descriptions of the PMS structure of C.mmp (a multiprocessor built from minicomputers) and its major components. An overview of its operating system, Hydra, is provided with emphasis on those sections most concerned with and influenced by the architecture. The project is also discussed in terms of performance, reliability, programming methodologies, and problems encountered.

In 1971 the Computer Science Department at Carnegie-Mellon University (CMU) undertook a project to construct C.mmp (Computer.multi-mini-processor), a relatively large-scale multiprocessor, from minicomputers. A number of project goals and criteria influenced the design:

- Minicomputers would be used as the processing elements of a multiprocessor that would support a general-purpose, time-shared environment.
- The machine would be symmetric: there would be no master-slave relation among the processors.
- A large address space would be provided.
- As much commercially available hardware as possible would be used.

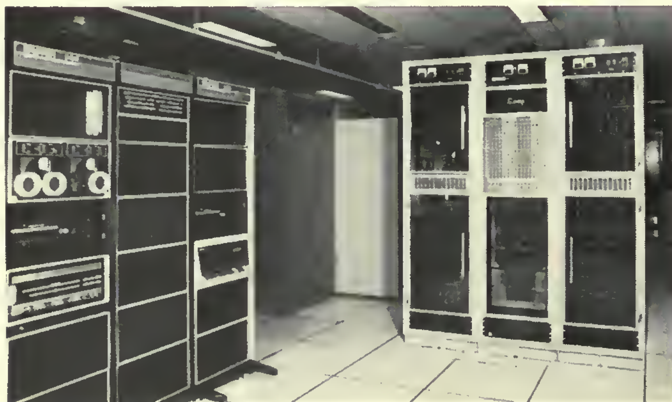


Fig. 1. The C.mmp multiprocessor.

To provide the necessary programming environment, a novel operating system was proposed, its principal component being its kernel, Hydra [Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack, 1974; Wulf, Levin, and Pierson, 1975]. The following criteria were used in designing the operating system:

- Separation of policy and mechanism: a kernel of mechanisms of "universal applicability" would be created from which varying policies could be implemented.
- A capability-based protection system and an object-oriented virtual memory would provide support for data abstraction; it would be extensible to user-defined data-types.
- The software would exploit the existence of multiple copies of many hardware elements for reliability.
- The structure of the system would be nonhierarchical.
- The system would be able to run for extended periods with no human operator.

The resulting C.mmp/Hydra-system has been completed and has met these goals. It has been running as a general departmental resource since mid-1975, supporting a time-shared user community as well as large-scale computing tasks, such as speech: understanding systems.

Table 1 summarizes the basic hardware and performance of C.mmp.

The Hardware: C.mmp

C.mmp is an asynchronous, multiple–instruction stream, multiple–data stream (MIMD) multiprocessor. To achieve the goal of symmetry, the processors and primary memory (Mp) are connect-

Table 1 C.mmp Hardware Summary

Structure	Symmetric, central cross-point-connected MIMD multiprocessor allowing up to 16 Pc's and 16 memories.
Processors	PDP-11 models 11/20 or 11/40, in any mix. A 16-Pc configuration of 11/20's and 11/40's was built. Eleven 11/40 models are currently in use.
Shared memory	32-Mbyte total shared address space. 2.7 Mbyte implemented using both core and MOS.
Secondary storage	700 Mbyte total moving-head disks. 6 Mbyte total fixed-head paging disks.
Performance	4.3 MIPS for 11/40 configuration, 3.0 MIPS for current 11/40 configuration. 26.3 $\times$ 10^6 references/s total memory bandwidth.

ed by a central cross-point switch. Before detailed design began, this structure was extensively studied by simulation and analytic models [Bhandarkar, 1972; Strecker, 1971], and it was determined that a 16 × 16 cross-point switch could be optimal, given the available technology. The TTL and Schottky TTL logic families were used for the switch and the relocation hardware because only they offered a fair range of MSI components. MSI components in the faster ECL logic were not available at the time. Essentially all of C.mmp is built with 1971–1972 technology, although some of the more recent additions use MOS LSI.

The Digital Equipment Corporation PDP-11 was chosen for the processors (Pc's) primarily because of its Unibus architecture. The Unibus allowed easy interfacing to the shared memory and kept the Pc modifications minimal. A further advantage of the Unibus was that it allowed DMA transfers to use relative, rather than physical, addresses because all addresses on the Unibus can be mapped in a uniform way by the relocation scheme, which will be described in detail. Therefore, the peripheral devices would need no modification to access the 25-bit shared memory address, even though they generate only the standard 18-bit Unibus address.

The following descriptions are primarily architectural, although some internal algorithms are described. For implementation detail, consult Fuller and Harbison [1978].

1.1 The PMS Structure

Figure 2 shows the PMS structure as of early 1979.[1] There are 16 processor ports and 16 memory ports in the cross-point switch (Smultiport, or Smp). The Pc's are slightly modified PDP-11/20 and PDP-11/40 processors, each connected to all the memories by Smp via the relocation unit (Dmap). The Pc's are further interconnected by an *interprocessor bus* (IP-bus), which provides basic control functions such as *start, halt*, and three levels of *interprocessor* interrupt (IPI), as well as the broadcasting of a 60-bit nonrepeating clock value used for interval timing and unique name generation. Note that this clock does *not* synchronize the internal operation of the processors.

C.mmp was constructed in several major stages: four prototype switches (1 × 1, 1 × 2, 2 × 2, 4 × 4), the full 16 × 16 switch with five 11/20's as processors, and finally the 16 × 16 Smp with a full processor complement of sixteen Pc's: five 11/20's and eleven 11/40's. The 16 memory ports were initially configured with the 1.4 Mbyte of core memory, and a similar amount of MOS memory was added later.

In early 1977 the Pc modifications for the 11/40 were completed, and by June 1977, C.mmp itself was completed by adding eleven 11/40's to the existing five 11/20's. Any mix of these two Pc models is possible. The desire to exploit a writable control store included in the 11/40 modifications, and performance measurements indicating that symmetry in processor speed is desirable,[2] led to exclusion of the 11/20's in early 1978, leaving the eleven 11/40's as the total Pc complement.

In the original PMS design [Wulf and Bell, 1972], a second cross-point switch was included to connect peripheral devices to any Pc's Unibus. For reasons of economy, this switch was never built and peripherals were assigned to specific Unibuses. I/O requests are mapped from requesting processors to the processor controlling the device via an IPI and a simple per-Pc queuing system in the operating-system kernel. The lack of the second cross-point switch has not been detrimental to the system.

1.2 Shared Memory Access

Access to shared primary memory (Mp) is performed in two stages: relocation of the 18-bit processor-generated address into a 25-bit address space, and resolution of contention in accessing that memory location. These jobs are performed by the *relocation unit* Dmap and the cross-point switch Smp, respectively.

1.2.1 The Relocation Mechanism: Dmap Dmap resides on the Unibus of each Pc and generally appears as a peripheral device, intercepting and mapping most addresses as they are placed on the Unibus. The planned, but not implemented, 2 Kbyte processor cache memory (Mcache) would interface to the Pc through Dmap.

Dmap divides the 32-Mbyte address space into thirty-two 8-Kbyte directly addressable pages that may be physically placed anywhere in shared memory. There are four address spaces, specified by 2 bits in the *processor status word* (PS). Therefore, four sets of eight address-mapping registers are provided in each relocation unit. To allow communication between address spaces without explicit addressing changes, the stack page is common to all four spaces.

The four address spaces are the heart of the memory protection mechanism: in only one space (1,1 in the PS space bits) are the relocation registers and the PS directly addressable. Since this page is used exclusively by the Hydra kernel [Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack, 1974], protecting the PS from indirect changes (see Sec. 1.5 of this chapter) guarantees

[1]Although shown in Fig. 2 to indicate its place in the architecture, only a prototype of Mcache was implemented.

[2]Many parallel decompositions of algorithms require that all processes synchronize between steps of computation. If some processes are running on slower Pc's, the processes executing on faster Pc's waste time waiting for the slower Pc's processes to report completion. The effect is like a convoy: all ships move at the speed of the slowest. See Sec. 3.1.2 of this chapter and Fig. 7 for a measurement of this effect.

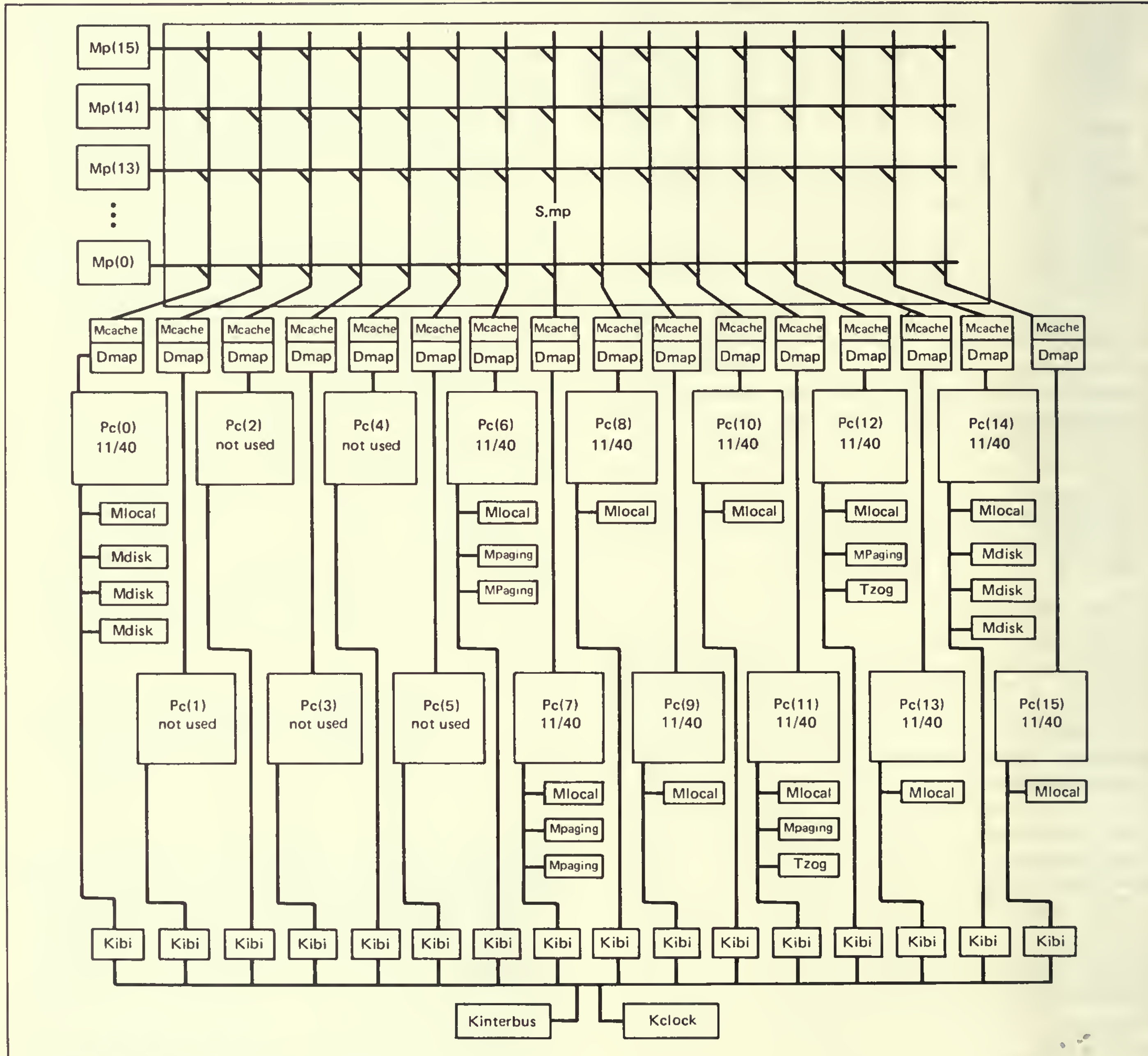


Fig. 2. The PMS structure of C.mmp.

that no addressability changes may be made without the approval of the operating system. All entries to the kernel, whether by interrupt or user request, force the assertion of both space bits.

To allow direct addressability, two of the relocation registers in (1,1) space are disabled, one each for the Mlocal page and the peripheral device control-register page. With these registers disabled, addresses that would normally be mapped are passed along the Unibus unchanged to be received by the addressed

memory or register location. Since the registers of Dmap are given addresses in the control-register page, they are always addressable by Hydra.

As illustrated in Fig. 3, the Dmap intercepts the 18-bit UNIBUS addresses (16-bit words plus the two space bits) and converts them in the following manner: the three high-order bits of the 16-bit word select a register from the bank specified by the space bits. The contents of the register provide a 12-bit page-frame number; the remaining 13 bits from the address word are the displacement within that page. The two are concatenated to form the 25-bit shared memory address. The 13-bit displacement gives an 8-Kbyte page size. This transparent mapping is performed for all shared memory accesses. In addition to the 12 page-frame bits, there are 4 bits in each relocation register used for control. The first three are designated *no page loaded*, *write-protected*, and *written-into*, and the fourth bit controls whether values from the page may be stored in Mcache.

After the 25-bit address is generated, Mcache is checked to see if the data are already available. If the access is a read cycle and the datum is in Mcache, the datum is immediately returned, bypassing shared memory. Although Mcache is a write-through design, only read-only data are cached, because the cache/Pc PMS structure allows multiple, and possibly different, copies of a datum. However, since approximately 70 percent of the memory accesses are to code rather than data, the read-only requirement is expected to produce a high "hit" rate for pure code programs [Fuller and Harbison, 1978].

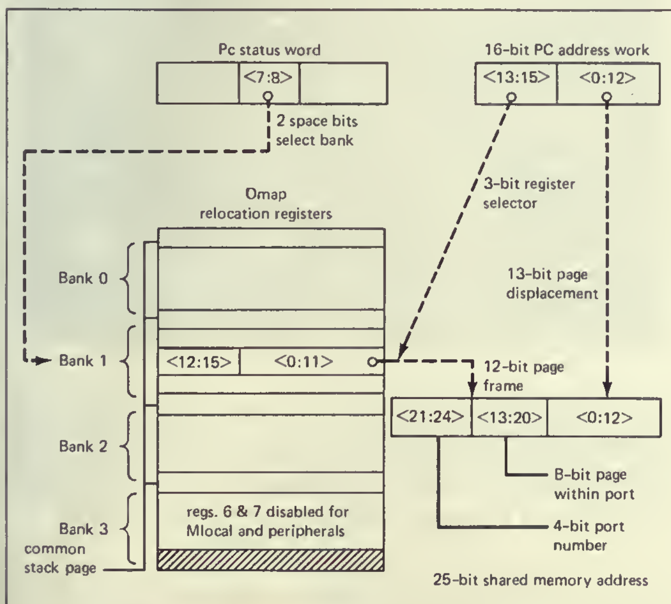


Fig. 3. C.mmp address relocation.

The internal characteristics of Mcache are:

Capacity	2 Kbyte
Block size	2 bytes (one word stored or returned per access)
Set size	1

Although only a single prototype of Mcache was built, it is estimated that it would save 50 percent of the time for a read cycle if the data were in the cache. However, it is important to realize that the motivation for including a cache on each processor was to reduce memory contention rather than directly provide fast memory. The effects of not having the caches will be discussed in Sec. 3.1.4 of this chapter.

Parity for both data and the 25-bit address is generated by the Dmap interface to the bus from the switch. The address parity is checked at the switch interface. If the check fails, the request is aborted and the processor interrupted. Data parity is not checked until the data are read from memory and returned to a Dmap. The fact that data parity is checked only by Dmap and not at any other point either in the cross-point switch or in the memory modules themselves has probably contributed to the reliability problems due to parity errors (see Sec. 3.3.1). Separate parity bits are maintained for both bytes of the word: one byte is given odd parity, the other, even. This detects words of all 1s or all 0s, both of which are common results of transient timing errors.

1.2.2 The Cross-Point Switch: Smp Smp routes the 25-bit address request to the memory port specified by the high-order four bits of the address. A port is requested by setting the bit corresponding to the requesting Pc in the port's *request register*. Contention for the port is resolved by periodically gating the request register into a second register, the *queue register*, which is left-shifted as the port becomes available. The shifting creates a priority-ordered queue: as a 1 bit is shifted out, the corresponding Pc is granted access to the port. Processor 15 is assigned the high-order bit and processor 0 the low-order bit, defining the priority. When the queue register is 0, all requests have been satisfied. The request register is again gated into the queue register and cleared, and a new cycle begins. A second request for the same port by a processor must enter via the request register; hence equality of service among the Pc's is maintained. The two-level request mechanism obscures the internal queue's priority ordering to the point that it is of virtually no importance outside the switch, preserving the symmetry of Smp.[1] The switch's maximum concurrency (16 independent paths) is achieved if all Pc's request different ports.

[1] In the worst case, in which all Pc's repeatedly access the same port, the lowest-priority Pc suffers a 50 percent memory access time degradation, but since this situation is extremely rare in practice, the effect is negligible [McGehearty, 1980].

The centralized and symmetric design of Smp makes the cost of memory access equal for all Pc's. Including address translation, switch overhead (no contention), and round-trip cable delay, the cost is about 1 μs. Although high by today's standards, more than equal to the access time of the memory, it has not proved prohibitive, or even annoying. The memory connected to Smp permits a maximum total bandwidth of 26.3×10^6 memory references per second, a value well matched to the speed of the 1971-vintage processors (see Table 2).

Smp was designed to allow partitioning of the system into smaller units. Each of the 256 cross-points has a switch that may be used to manually enable or disable it. These switches, plus a global cross-point set switch, set the flip-flops that control the individual cross-points. Now disconnected for reliability and software security, there was a program interface that allowed setting of the cross-point configuration from Pc 0.

The ability to partition the system was originally intended to allow multiple versions of the operating system to coexist. However, funds were not available to provide sufficient primary and secondary store to allow simultaneous execution of multiple copies of Hydra. Currently, the principal use of the manual cross-point enable switches is to disconnect faulty hardware elements. A Pc and a single memory port are sometimes partitioned out of the system to allow maintenance to proceed concurrently with normal operations.

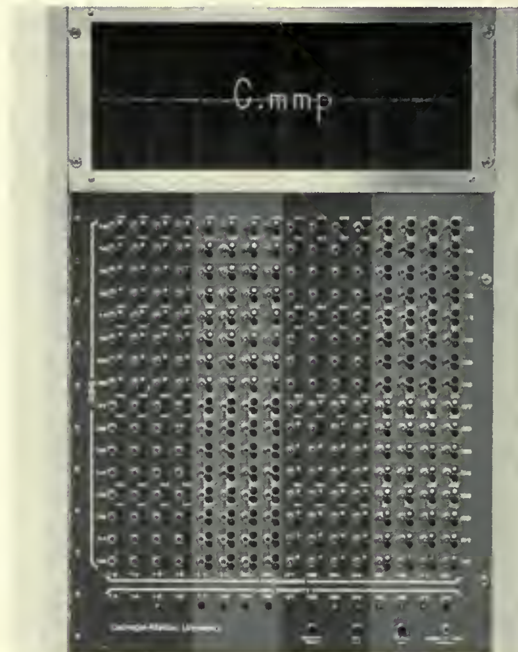
1.3 Primary Memory

The current complement of primary memory is 2.7 megabytes of mixed technology: eleven ports containing 1.4 megabytes of core memory and five ports with 1.3 megabytes of MOS memory. Technologies are not mixed within a memory port.

The memory port control of Smp permits each port to be interleaved in as many ways as there are independently driven memory modules. Interleaving is specified by the page number, bits 13 to 20 of the 25-bit address (see Fig. 2). C.mmp's core memories are 16-Kbyte modules, and there are eight independently driven modules per port, allowing eight-way interleaving.

Table 2 Shared Memory Characteristics

Core memory	250-ns access, 650-ns cycle time 16-Kbyte module size, 8 modules per port 8-way interleaved within a port 1.71×10^6 references/s per port maximum bandwidth
MOS memory	330-ns access, 450-ns cycle time 65-Kbyte module size, 4 modules per port No interleaving 1.49×10^6 references/s per port maximum bandwidth



(a)

(b)

Fig. 4. The crosspoint switch. (*a*) The crosspoint display panel with the system partitioned into two disjoint 8 x 8 machines. (*b*) A detail of the display panel. Three Pc's are selectively permitted access to ports 2, 3, and 4 as shown by the crosspoint enable (CPE) lights. Two of the Pc's are actively accessing memory (ACT lights).

The MOS memory has four 65-Kbyte modules per port. However, they are not independent, having only one refresh control board,

and so are not interleaved. Ports can have up to 256 pages, or 2 megabytes, of memory. Table 2 provides specifications and measurements of the memories.

Each Pc also has 8 Kbyte of local (nonshared) core memory (Mlocal).

1.4 The Interprocessor Bus

The IP-bus provides a common clock as well as interprocessor control. These two logically and functionally separate features use separate data paths, although they share a common control (Kinterbus). Each processor has an *interbus interface* (Kibi) that defines the processor's bus address and makes available the bus functions to the software.

The first function is to continuously broadcast the 60-bit, 250-KHz Kclock. This is done by multiplexing the clock value onto a 16-bit-wide data path in four time periods, low-order bits first. Any Kibi requesting a Kclock read waits for the initial time period and then buffers the four transmissions in four local holding registers available to the software. Clock values are often used for unique names [Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack, 1974; Wulf, Levin, and Pierson, 1975], and so the otherwise unused high-order four bits of the fourth local register are set to the reader's Pc number to ensure uniqueness when any number of Kibi's read the bus simultaneously.

Each Kibi has a countdown register for interval timing. It may be initialized to a nonzero value by the program, and it is decremented by 1 every 16 μs (timing supplied by Kclock). The Pc is interrupted when the register reaches zero.

The second bus function is the interprocessor interrupt and control mechanism. Each Pc may interrupt, halt, continue, or start any other Pc, including itself. Each Kibi has a 16-bit register for each of the control operations. The operations are invoked by setting the bit(s) corresponding to the processor(s) to be controlled in the appropriate register. Setting the *i*th bit invokes the operation on Pc(*i*). A second 16-bit-wide data path is eight-way–time-multiplexed, each control operation being assigned a time period. As the appropriate period arrives, each Kibi ORs its control operation register onto the bus and clears the register. Synchronization of bus access, as well as operation specification, is accomplished by the multiplexed time periods. The Kibi also inspects the bus to see if the specified operation is being invoked on its processor; if so, the requested action is performed. Although eight time periods are available, only six are used: three priority levels of IPI, halt, continue, and start; the remaining two are ignored.

Each Kibi provides a manual switch register that defines the set of Pc's that the host Pc may interrupt or control. As with the control operation registers, setting switch *i* permits the Pc to invoke IP-bus functions on Pc(*i*). These registers, one per processor, are used with the manual cross-point enable switches

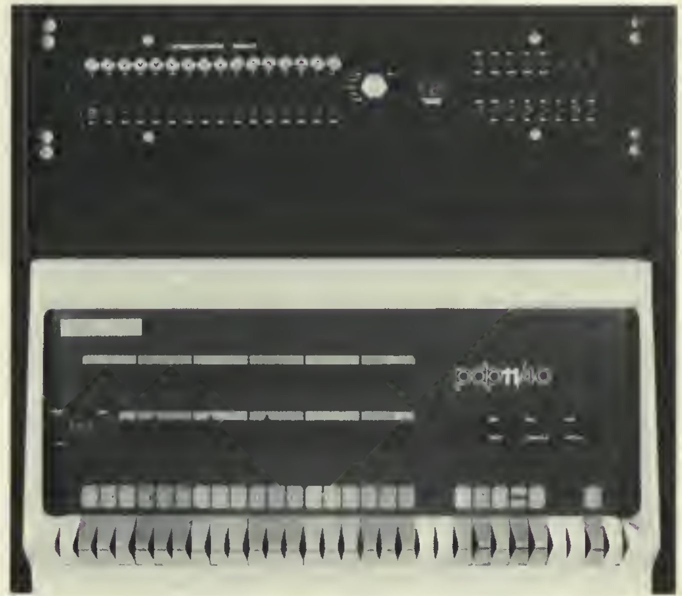


Fig. 5. A typical C.mmp processor with its Kibi.

to partition the system. A 16-bit LED display register is also provided to selectively display the four words of Kclock or the interval-timing counter and its control register.

1.5 Pc Modifications

The PDP-11's used on C.mmp were slightly modified to provide software protection and make the Pc's compatible with a multiprogramming environment. Also, a writable microstore was added to the 11/40's. The actual modifications were similar for both PDP-11 models; however, their implementations were quite different because of the differing internal implementations of the two models. In neither case were the changes extensive. Certain instructions were made privileged to ensure the integrity of the system software. In particular, HALT, WAIT, and RESET were prohibited from user programs. Since the processor status word (PS) controls the relocation address space of the executing program (and hence memory protection), two instructions which may modify it from user space were also prohibited: RTI (return from interrupt) and RTT (return from trap). Both of these instructions load the PS from the stack. Since they are sometimes used in subroutine calling sequences, they are trapped and emulated by the kernel for user executions—after an appropriate checking of the new PS to be loaded.

Because the operating system must leave some context information on the stack during protected procedure calls [Cohen and Jefferson, 1975; Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack, 1974], address bounds checking was added to the *stack*

pointer register, R6. Stack overflow protection existed, but it was necessary to augment it with underflowing checking. The *stack underflow register* (SUR) prohibits all accesses to the stack, page at addresses higher than its contents.[1] This protection extends to all accesses, whether by stack operations or direct addressing, thus protecting the previous context information. Additionally, R6 is constrained so that its contents always lie in the stack page (page 0) of Dmap.

Because of the difficulty of modifying a processor, the stack underflow register and the comparison circuitry were physically placed on one of the relocation unit boards. This remote placement compounded the timing difficulties of adding stack-limit checking to the processors. Having to protect the PS by disallowing user execution of RTI and RTT increased the perturbation of stack-operation timing. Unfortunately both of these modifications were necessary to ensure safe operation of a multiprocessing, multiuser operating system.

1.6 Writable Microstore

The PDP-11/40 is implemented via a horizontal microprocessor [DEC, 1972] with provision for extended control store to implement various instruction-set options. At CMU, a writable control store was developed in place of the standard extensions [Fuller, Almes, Broadley, Forgy, Karlton, Lesser, and Teter, 1976]. The writable store contains 1,024 eighty-bit words, a general mask-shift unit used for field extraction and data manipulation at the microprogram level, and a microprogram subroutine facility.

No such extension was possible for the 11/20, since it is not a microprogrammed processor. This asymmetry in the configuration was a major reason for the removal of the 11/20's.

1.7 Peripheral Devices

Peripheral devices on C.mmp are standard PDP-11 Unibus-interface devices; no modifications are required. Two of the device types are unique: the zero-latency paging disks and the graphic displays. The paging disks have 8 Kbyte per track, which exactly matches C.mmp's page size. Their controllers achieve zero latency by continuously monitoring the position of the disk under the fixed heads and, for full track transfers, can start the transfer at any 16-word sector boundary, calculating the proper displacement into the page. As the disk turns, the memory address is "wrapped around" when the start of track is reached.

The graphic displays are a CMU-designed and -built vector display [Rubin, Guggenheim, and Bihary, 1978]. The two on C.mmp are equipped with a transparent touch screen in front of the CRT display for specialized man-machine interaction studies in the ZOG data base management project [Robertson and Ramakrishna, 1977].

Table 3 summarizes the major devices and is an indication of the capabilities of the machine.

[1]In the PDP-11 instruction set, stacks grow from higher to lower addresses.

2. The Software: Hydra

A discussion of C.mmp would not be complete without an introduction to its unique operating system, Hydra. Hydra provides two basic mechanisms: (1) process creation and scheduling and (2) a capability-protected, object-oriented virtual memory system for date abstraction. In this section, emphasis will be placed on those features of the Hydra kernel most related to the multiprocessor architecture.

2.1 Processes, Scheduling, and Control

The features of Hydra most directly influenced by the architecture are process scheduling and control. The heart of Hydra's multiprocess, multiprocessor scheduler is the Kernel Multiprogramming System (KMPS). This system also implements several of the process control functions, including the synchronization primitives.

2.1.1 Processes and Scheduling: KMPS In Hydra, the unit of scheduling is the *process*. Process scheduling is done in two phases: long-term (job selection) and short-term (context-swap frequency). Perhaps nowhere else in the kernel is the notion of policy-mechanism separation so clearly employed as in this two-level scheduler [Levin, Cohen, Corwin, Pollack, and Wulf, 1975].

KMPS provides the basic process creation and scheduling mechanisms as a parameterized short-term scheduler. It is driven, in turn, by one or more long-term schedulers. These schedulers, known as *policy modules* (PMs), are implemented as user-level programs and provide independent scheduling policies for different job streams, such as timesharing and batch.

Table 3 Major Devices on C.mmp

Quantity	*Device type*
3	200-Mbyte moving-head disks, 3330-type
2	40-Mbyte moving-head disks, 2314-type
2	20-Mbyte moving-head disks, 2314-type
6	1-Mbyte fixed-head, zero-latency paging disks
2	Vector graphic display terminal with touch screens
1	600 l/min line printer
1	9-track magnetic tape drive
1	Interface to ARPANET
n	Assorted local terminal interfaces

On account of the symmetry of the architecture, processes usually need not be bound to specific processors. KMPS schedules among processors as though the Pc's were merely a resource pool. The PMs need not be concerned with the multiprocessor aspects of scheduling. A PM simply supplies KMPS with a stream of processes to be run; KMPS will make the necessary multiprocessor scheduling considerations.

KMPS schedules according to four basic parameters supplied by the PMs for each process:

Process priority	The process's relative priority among the set of processes controlled by KMPS
Time quantum	Maximum execution duration, composed of a time-slice length and number of slices
Processor mask	A bit mask of permissible Pc's for the process; normally set to indicate any Pc
Maximum page set size	The maximum number of pages that the process may have resident in Mp at any given time

When a process is started, the four parameters are set and KMPS places it on the *feasible* list, a list of runnable processes. When selected from this list, a process may execute until it blocks, completes its time quantum, or is preempted by a higher-priority process. If preempted, the process is returned to the feasible list and waits until resources are again available at its priority. KMPS reconsiders its scheduling at the end of each time slice on any Pc; all Pc's execute KMPS asynchronously. When a process consumes its time quantum, it is returned to its controlling PM for reconsideration of long-term scheduling.

The basic KMPS mechanisms for scheduling and multiplexing the processes onto Pc's are quite straightforward: First, the highest-priority is chosen from the feasible list. Then, according to the process's processor mask, the highest-priority Pc is chosen, and the process is enqueued for that Pc. The Pc is then sent an IPI instructing it to reconsider its scheduling. If the incoming process is of higher priority than the one currently running, a context swap to the new process takes place and the previous process is returned to the feasible list. If the incoming process is not of higher priority, it is returned to the feasible list and no rescheduling takes place. Allowing the selected processor to make the scheduling decision at a time of its choice (controlled by Pc interrupt priority) helps to eliminate race conditions that would otherwise be rampant because of the asynchronous nature of C.mmp.

The scheduling mechanisms are quite efficient, since only half the mechanism need be invoked for most operations. Usually either the process or the Pc is known. For example, at the end of a time slice the Pc is known to be free and all that is needed is to identify the highest-priority process that it may execute. Similarly, a blocked process that is awakened only requires that a processor be assigned. The full mechanism is needed only when a new process is introduced into KMPS control. An additional mechanism allows Pc's with heavy DMA or interrupt traffic to be shielded from computational burdens by assigning them a lower priority. High-priority Pc's that have become idle can "steal" processes from lower-priority Pc's, freeing them for I/O duties. This mechanism is important in reducing overrun errors (see 3.3.1).

With these mechanisms, KMPS is capable of controlling a large number of processes; the system routinely runs with more than 100 processes without inordinate overhead.

2.1.2 Synchronization One of the most crucial functions of an asynchronous multiprocessor is its ability to synchronize independent instruction streams when required. Hydra uses, and provides at user level, a number of synchronization mechanisms. Most basic of these is the spin lock, implemented by continuous polling of a shared memory location. Because of the memory contention generated, spin locks are generally undesirable and are avoided within the kernel. However, because the fast mechanisms of the kernel are not available at user level, spin locks are sometimes useful for brief critical sections in user programs.

The most important synchronization mechanisms are the KMPS lock and the two forms of semaphore [Dijkstra, 1968a] implemented by Hydra. Another mechanism, based on message passing, is discussed in the next section. While these mechanisms are semantically equivalent, they differ widely in implementation and timing characteristics. The choice of mechanisms is dictated by both synchronization context and performance considerations.

The KMPS lock is a low-level, mutual-exclusion primitive operating below the process level. It is the logical equivalent of a spin lock, but its implementation uses interprocessor interrupts to avoid the memory contention inherent in continuous polling. The use of KMPS locks is restricted to places where context swap is not allowed, such as in interrupt routines.

A KMPS lock is implemented with two counters and a bit mask of waiting Pc's. When a lock request is made, the lock counter is indivisibly decremented (from 1) and tested. If the result is 0, the requesting Pc has control of the critical section. Otherwise, the Pc must wait. In this case, the Pc places its bit in the waiting processor mask and executes a WAIT FOR INTERRUPT instruction, idling the Pc. When a Pc unlocks a lock, it increments the lock counter, sets the second counter (sublock) to 1, and sends the highest-level IPI to all Pc's in the wait mask.

The blocked Pc's, upon receipt of the interrupt, resume execution and contend for the sublock. One, randomly determined, will see that its decrement of the sublock field has resulted in 0 and will remove its bit from the mask and assume control of

the critical section. The others resume waiting. By allowing the sublock to be reset on each unlock operation, the lock counter contains the number of processors blocked (negated) while the lock is locked. This information is used in consistency checks that detect either incorrect lock addresses or damaged locks.

The advantages of this apparently complex system are twofold: it is extremely cheap in the nonblocking case (most frequent), and there are no memory cycles consumed in blocking, although the Pc is unavailable. The performance of this mechanism is excellent and will be discussed in a following section.

Semaphores differ from locks in two ways: their counters may have large values, and since they are process-level primitives, blocked processes are rescheduled. Each semaphore maintains a queue of blocked processes that will be rescheduled in the order that they have blocked.

Two forms are supported: one internal to Hydra (kernel semaphore) and one for user-level programs (PM semaphore). The difference is (conceptually) in their behavior when blocking. If a process must block on a kernel semaphore, a token for the process is appended to a queue within the semaphore and the Pc selects a new process from the KMPS feasible list. In particular, the pages of the blocked process remain core-resident.

Blocking on a PM semaphore is more complex. Not only is a token for the process enqueued, but a scheduling decision to swap the pages of the process must also be made. This decision is delayed for a period (currently 500 ms, a parameter controlled by the PM), so that if the critical section is freed during this time, the process may possibly continue. In this case, the behavior is much like the faster kernel semaphore and averts considerable paging overhead. If during the delay the process cannot continue, it is returned to the PM for the duration of its blocked period and its pages become eligible for swapping. Although this mechanism pays a penalty of potential paging overhead, it ensures that a deadlock in user code does not result in a kernel deadlock. Upon receipt of the signal that the process may enter the critical section, the PM will again consider it for long-term scheduling and order it restarted by KMPS.

2.1.3 Interprocess Communication A variety of hardware and software communication mechanisms are available within C.mmp/Hydra. The hardware provides two: First, and most basic, is sharing memory, used extensively by both kernel and user-level programs. Second is the IP-bus control functions, which are used strictly within the kernel. The three IPI levels are used for scheduling, interprocessor I/O request queuing, and synchronization. The IP bus halt and start functions are used during system initialization and by a monitoring Pc to regain control of a Pc lost through serious error.

Hydra provides two software mechanisms: an interprocess interrupt (analogous to the IP bus interrupt for Pc's) and a message facility. The KMPS *control* function allows one process to interrupt another. Control interrupt entries are made at specified points associated with each process. Each process also has a control mask associated with it; the process sending a control function supplies a similar mask. A nonzero intersection of the masks causes the interrupt to be taken.[1] Depending on the interrupt, additional data may be available in certain predefined stack addresses [Newcomer, Cohen, Jefferson, Lane, Levin, Pollack, and Wulf, 1976]. A similar function, *desynch*, can be used to free a process blocked on a PM semaphore or while waiting for a message. In this case, an exception return is made from the blocking kernel call.

The Hydra Port System provides a general message facility that can be used for user-level interprocess communications and synchronization [Newcomer, Cohen, Jefferson, Lane, Levin, Pollack, and Wulf, 1976]. Messages are sent to and received by *ports*,[2] which may be interconnected via unidirectional links between an output channel of one port and an input channel of another. The messages are typed and may contain both data and a single capability (discussed in the next section). The basic port operations are SEND, RECEIVE, and an RSVP function that requests a reply to the message sent. Because the memory protection system provides protection only on a per-page basis, messages, which are always smaller than a page, must be created within the kernel and therefore are not directly addressable by user programs. Additional mechanisms, necessary only because of the memory protection limitations, are provided for creation of messages and copying of their contents.

The RECEIVE operation may block until a message is received by a port. Since ports, not processes, are connected, blocking provides a way to synchronize a dynamically changing set of cooperating processes in a producer-consumer relationship. No process requires knowledge of the number, role, or memory of the other processes: it knows only of its connection to a shared port and the operation it is to perform.

The Port System also provides a uniform user-level interface to the I/O system. Devices appear as ports, and requests are entered by sending an appropriately formatted message to the device. The fact that devices are physically connected to specific Pc's is completely obscured, and the common interface allows easy interchange of similar devices.

2.2 Protection and Data Abstraction

Although the protection and data abstraction mechanisms of Hydra are not dependent upon the architecture, the following

[1]Interrupting a user program must be simulated by Hydra to protect the PS. Exception interrupts, such as attempting to access nonexistent memory, have entry points associated with each process similar to the entry for control interrupts. When entry is made, the stack is loaded by Hydra with the (PC,PS) pair to simulate an interrupt to the entry point.

[2]Not to be confused with Smp memory ports.

brief introduction to these mechanisms is presented, since they were among the principal design criteria for the operating system. Detailed discussions may be found in the Hydra monograph Wulf, Levin, and Harbison, 1980] and also in Cohen and Jefferson [1975], Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack [1974], and Newcomer, Cohen, Jefferson, Lane, Levin, Pollack, and Wulf [1976].

In Hydra, all data are encapsulated in *objects*, which may only be accessed via *capabilities*. The set of all objects is known as the Global Symbol Table (GST, pronounced *ghost*). Capabilities have a varying set of access and operation *rights* that are automatically checked whenever a capability is used to name an object. If a capability has insufficient rights for the requested access or operation, a signal is returned to the caller pointing out the protection violation. Hydra provides a set of 16 generic rights that are interpreted uniformly for all object types. An additional eight rights, the auxiliary rights, are available for each object type, and their interpretation is dependent on their type. Sharing of objects is permitted by copying capabilities for the object, possibly with the access rights restricted to limit authority.

Objects have a unique name (generated from the 60-bit clock), a type, and optionally a *data part* and list of capabilities (*C-list*). The data part allows storage of a limited amount of data (4,000 bytes). The C-list allows an object to contain up to 250 references to other objects. General graph structures may be built via these capability references. The protection mechanism is not hierarchical and may be used to protect structures with arbitrary interconnections. Objects may be referenced via a *path* of capabilities in the C-lists of other objects (if all capabilities along the path have sufficient rights).

The representations of both capabilities and objects are never directly manipulated by user-level programs; all representation knowledge is the domain of the kernel.

Nearly everything is represented as an object: processes, pages, semaphores, I/O devices, ports, and a great variety of other types. Every executing program has a basic list of capabilities known as its *Local Name Space* (LNS). The LNS and all objects reachable by paths rooted in the LNS are the instantaneous protection domain of the program. To prevent forgeries, objects may be referenced only by such paths; they are never directly referenced by name. An LNS typically contains capabilities for its code and data pages plus capabilities for any other objects that the program must manipulate.

Protected procedure calls switch protection domains. All programs are represented by *procedure objects*, which have C-lists containing capabilities for code pages, data pages, and *parameter templates*, as well as any other objects required. A procedure may have capability parameters in the same sense that a subroutine has address reference parameters. When called, the procedure's C-list and the actual parameters are merged into an LNS for the new protection domain. Procedure calls stack LNS's, so that calls may be nested or recursive. The templates specify the parameter's position in the LNS and the necessary *check rights* and type of the actual capability. If the check rights or type doesn't match, a protection violation is signaled and the call aborted. A parameter template may also specify *rights amplification* to add certain rights to a parameter capability. The amplified capabilities exist only in the LNS of the called procedure and are rarely, if ever, returned to the caller.

Two other forms of template are used. *Creation templates*, which specify the initial form and type of an object, are used by a common object creation routine to create an instance of a particular type. *Amplification templates* provide rights amplification outside the procedure call mechanism. These are not made generally available.

The kernel provides a small number of basic object types and a mechanism for creation of user-defined types. A new type is represented by a *TYPE object* that embodies the abstractions of a class of objects. The TYPE object specifies the representation of data in the new class and also the operations that may be performed on the object. Auxiliary rights may be defined to protect these operations. The code defining the representation manipulations of an operation is encapsulated as a procedure object and is stored in the C-list of the TYPE object. Generally, only these procedures may use rights amplification, either by template or in the procedure call, to gain sufficient rights to directly access the representation of the new type. To allow use of the new type, a creation template, made from the TYPE object, is made available as needed.

To invoke an operation on an instance of the new object type, Hydra provides a *typecall* mechanism similar to the protected procedure call. Performing a typecall on an object of the new type actually specifies a call on one of the procedures in the C-list of the TYPE object. A capability for an object of the specific type is passed, possibly with rights amplification, to the procedure implementing the desired operation. A different typecall is provided for each operation on an abstract data-type, and the index of the procedure in the type object is typically hidden in a macro or routine in the source language.

The typecall mechanism is used to implement all user-level subsystems in the Hydra operating system. For example, PMs create PM process objects to encapsulate PM scheduling data. For a detailed example of how typecall was used to create an extensible file system, see Almes and Robertson [1978].

3. The Hardware-Software Interaction: Performance, Programming Methods, Problems, and Reliability

Developing the operating system and implementing several large application programs has resulted in a considerable body of knowledge about how the architecture has interacted with the

software. Some expected problems, such as multiprocessor scheduling and synchronization, have been solved efficiently and effectively. Others, mostly unanticipated, have been difficult to solve or minimize, although in one case—reliability—the software methods developed are considered one of the project's major successes [Wulf and Harbison, 1978].

3.1 System Performance

The following sections present an overview of the performance of the C.mmp/Hydra system. Again, the emphasis is on the architecture and its effects. The data presented have been collected over a period of years and represent a number of different system configurations, since measurements were taken in parallel with hardware development. The concurrent measurement and construction unfortunately prevented simultaneous measurement of more than a subset of the potential 16 Pc's. To offset this, modeling results extending the measured data are presented where available.

To measure the system, a number of specialized tools were created. Two software tools were created to measure the system behavior in parallel execution. A *software tracer*, partially implemented in microcode, was built to selectively trace events such as kernel calls and object accesses on the entire set of executing Pc's. A *script driver* [McGehearty, 1980] provided a mechanism to impose a variable and repeatable synthetic load on the system and make timing measurements at the user level. A special *hardware monitor* [Swan, 1976] with its own host computer was developed to measure performance at the memory-access and instruction level on individual Pc's. The monitor's high-impedance probes, which were attached to the measured Pc's Unibus, allowed fine-grained measurements to be taken with insignificant perturbation of the Pc. Memory traffic in Smp was measured with an access counter that integrated accesses to all 16 memory ports.

3.1.1 An Application Example An artificial intelligence application, the Harpy Speech Understanding System [Lowerre, 1976], was implemented on C.mmp, among other machines. The system was extensively studied as an indicator of the performance and problems associated with large, complex tasks in the C.mmp/Hydra environment.

The following brief description of Harpy and its implementation on C.mmp is presented to aid understanding of the application and its measurements. The system recognizes speech from many speakers, although the recognizable utterances are restricted to a finite, task-constrained vocabulary. Knowledge about the task, grammar, and vocabulary is represented in a finite-state graph structure, one word of the vocabulary per node. Paths along interconnections between nodes represent acceptable sentences in the grammar. When an utterance is to be processed, the word nodes are replaced by networks containing representations of the phonemes (units of speech) for all pronunciations of the words.

After digitization, an utterance is examined by a probability-based heuristic search that compares each phoneme of the utterance to those in the nodes of the knowledge graph. As the search proceeds, a recognition tree of the most probable transitions in the graph is built. At the end, the utterance is identified by backtracking along the path of highest probability in the recognition tree.

The search was implemented in two phases, each executed by a set of cooperating processes. In the first phase, the possible transitions in the knowledge graph were calculated for the current phoneme of the utterance. The second phase performed a probability calculation for each transition identified in the first phase and discarded those of low probability. Steps of high probability were retained as the next level of the recognition tree. The processes were synchronized so that all performed the first phase, then all performed the second phase. This sequence was iterated until all phonemes of the utterance were processed. No process was allowed to continue to the next phase until all processes had completed the current phase.

To ensure that the measurements were indicative of the architecture, the number of processes was limited to the number of Pc's available at the time of measurement and the code and data were always resident in Mp. These precautions eliminated the effects of scheduling and paging. The measurements in Fig. 6*a* [Oleinick, 1979] were made with a 1,000-word vocabulary, representing a large search space and heavy compute load. The same 15 utterances were processed for each measurement.

Since versions of Harpy also exist for the PDP-10 (the KL10 model, a medium- to large-scale uniprocessor with 1.8 MIPS, and also an older KA10 model with 0.4 MIPS) and an 11/40 Unix system, some performance comparisons can be made with these systems. As Fig. 6*a* shows, C.mmp achieved better performance than KL10 with four Pc's on the 1,000-word vocabulary task. In comparison with the single 11/40 Unix system, shown in Fig. 6*b*, a single process on C.mmp required only slightly greater time to execute the task than did the uniprocessor, indicating that overhead is low in the parallel environment [Wulf and Harbison, 1978]. For the Unix and KA10 measurements, a small (37-word) vocabulary was used. For reference, measurements of the KL10's performance on the 37-word vocabulary task are also included in Fig. 6*b*.

The speedup gained by adding a Pc to Harpy was less than linear on account of underutilization of the processes [Oleinick, 1979]. Because of unequal allocation of work, the processes lost time waiting for the working processes to complete a phase of the search so that the next phase could begin. Considerable effort was invested in optimizing the allocation of work, and process utilization reached 64 percent, limited by the overhead necessary in partitioning the heuristic search. The partitioning overhead is

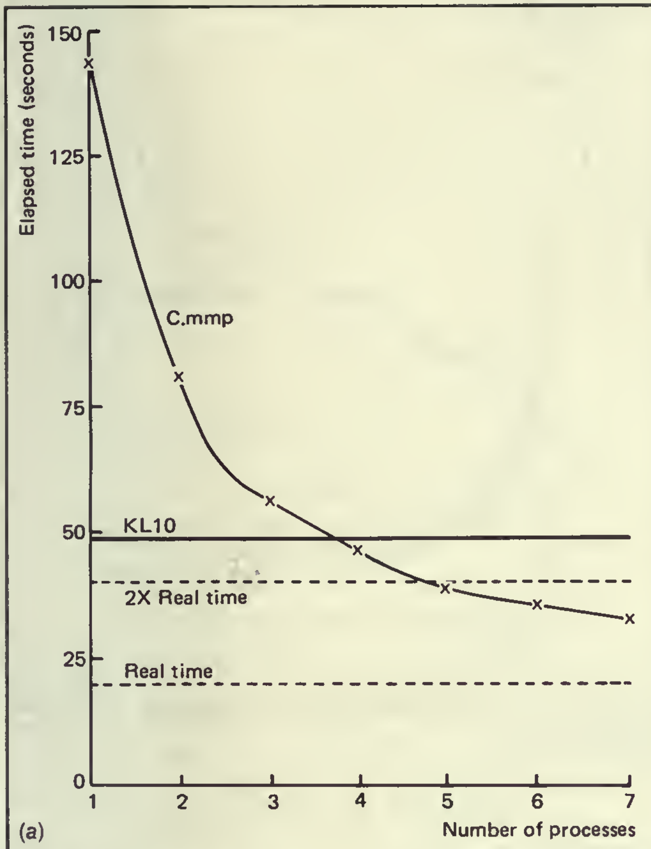
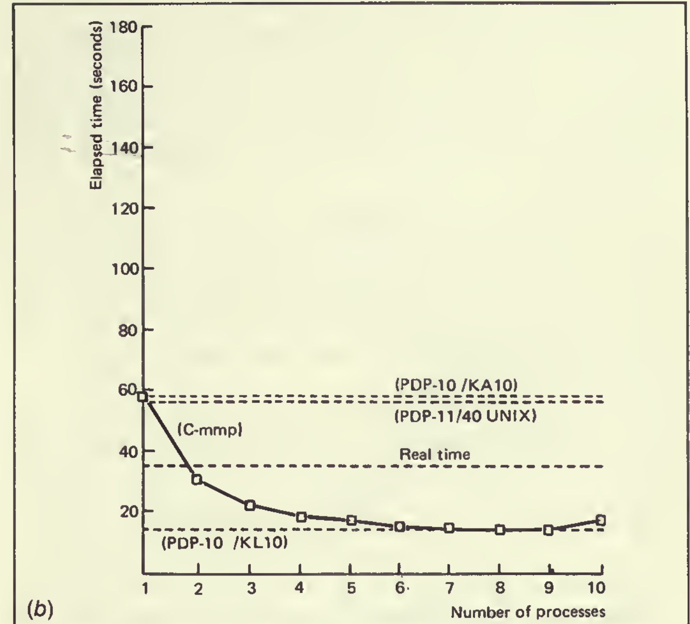


Fig. 6. Comparative performance of C.mmp with other machines on the Harpy Speech Understanding System. (a) C.mmp versus KL10 with a 1000-word vocabulary task.



(*b*) C.mmp compared with a 11/40 Unix System, a KA10, and a KL10 with a 37-word vocabulary task.

also responsible for the fact that C.mmp required seven Pc's to match KL10 performance on the 37-word vocabulary task but only four on the more computation-intensive 1,000-word task. Although this problem is seen to be in the decomposition of Harpy rather than specific to C.mmp/Hydra, it demonstrates a problem with multiprocessors and parallel decompositions: if a Pc must wait for all other processors to complete their task before proceeding to the next computational step, then speedup will be limited by the balance of work among the Pc's. This is similar to the problem of unequal execution speeds mentioned in 3.1.2.

3.1.2. Synchronization and Its Effects Studies of the synchronization mechanisms in Hydra indicated that the mechanisms themselves did not cause much overhead, although the methodology of use was critical. In a study of the KMPS lock, several benchmark programs each created 16 cooperating processes designed to cause varying frequencies of synchronization within the kernel [Marathe, 1977]. For these experiments, a 14-Pc configuration (including the 11/20's) was used so that scheduling 16 processes would ensure full Pc utilization. The measurements, taken with the hardware monitor to avoid any perturbation of the system, indicate that in the worst cases:

- Fewer than 10 percent of locking requests blocked.
- Less than 1 percent of execution time was lost in blocking.
- The duration of an average KMPS lock-protected critical section was less than 700 μs.

A model, verified by the hardware monitor measurements, predicted that if the KMPS lock mechanism were extended to a 48-Pc system, the time lost in blocking would be less than 4 percent [Marathe, 1977].

Although the lock-unlock code is highly efficient, the fact that so little time is lost in blocking is due primarily to the methodology of synchronization. The critical factors are association of locks with data structures rather than code segments, and choosing synchronization primitives on the basis of the duration of the critical section to be protected. By associating synchronization primitives with data structures, several processes may execute the code for a critical section without mutual interference, since they each lock different locks. Contention is limited to the degree of sharing of a

specific instance of a data structure. In Hydra, nearly every shared data structure has its own locking primitive. For example, each of the tens of thousands of objects in the GST has a KMPS semaphore for mutual exclusion. Also, some highly shared structures, e.g., the KMPS feasible list, are segmented to allow multiple locks and a higher degree of parallel access without contention.

Another advantage of associating the synchronization with data structures is that both the primitives and data structures may be dynamically created and destroyed as the load or growth of the system may dictate. The code remains unchanged during the lifetimes of the dynamically created data structures. Binding locks to code results in static structures that require programmer intervention for alteration.

The importance of choosing the appropriate primitive was shown by a study of a parallel root-finding algorithm [Oleinick and Fuller, 1978]. Figure 7 illustrates the difference between spin locks and PM semaphores. Because the average blocking time is short compared with the overhead of PM semaphores (at least 5 ms), the spin lock produced better performance by a factor of 2. For the PM semaphore curves, the e parameter is the delay time in milleseconds before a blocked process was returned to the PM. Note that zero delay ($e=0$, in PM0, an early PM) causes poor performance due to paging overhead (see Sec. 2.1.2). The degradation of performance caused by adding the ninth Pc is not due to the synchronization primitives, but is caused by the system configuration: eight 11/40's and three 11/20's. As soon as one 11/20 was used, the entire task force of processes slowed down, since all were forced to wait for the slowest to report completion.

The choice of primitive is equally important in Hydra. If the estimated average blocked period was greater than context-swap time, a KMPS semaphore was used; otherwise a KMPS lock was best. Measurements indicate that the average KMPS semaphore blocked period ran as high as 300 ms [Jain, 1978] because semaphores were used for signaling I/O event completion. Clearly, if locks, which do not release the Pc, had been used, the impact on performance would have been severe.

3.1.3 Scheduling The script driver was used to measure the combined performance of KMPS and the PM as it would be perceived by a terminal user, especially with respect to variation in response time [McGehearty, 1980]. The load placed on the system was controlled by both the number of jobs (terminal users) and the compute time required by each job. To minimize effects other than scheduling, several restrictions were placed on the synthetic job stream:

- All jobs were independent.
- All jobs executed at the same KMPS priority.
- Jobs made no accesses to the GST (which might cause I/O or contention for an object).

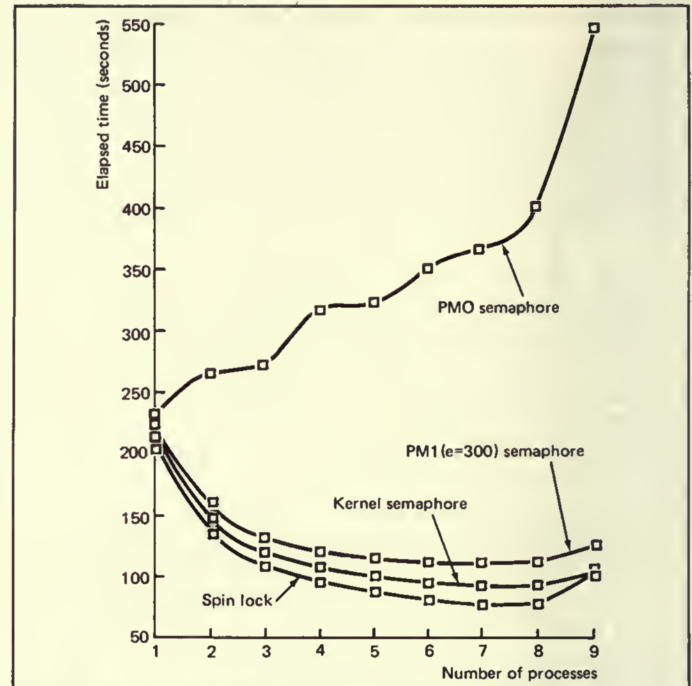


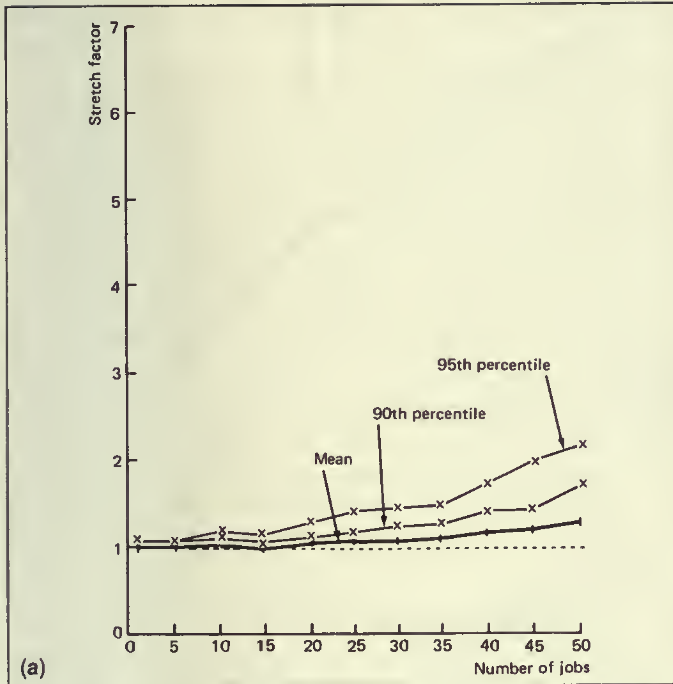
Fig. 7. Effect of different synchronization primitives on the root-finder program.

- All codes and data simultaneously fitted into Mp.

Timings were measured by the script process using the global time base, Kclock. For this experiment, a configuration of ten 11/40's was used. Nine Pc's executed the synthetic job stream and the tenth ran the script driver process (which was locked onto that Pc to prevent interference with the measurements).

To create a synthetic user load, the script driver jobs were assigned compute times in exponential distributions with mean times of 1, 5, and 10 corresponding to light, medium, and heavy loads. Each job waited 10 s between compute requests to simulate user response time. A total of 400 compute/wait cycles were executed for each set of jobs. Because of the varying compute time requests, the variation in response time was normalized for all processes by calculating a *stretch factor*, the response time divided by the requested compute time.

The variation of the stretch factor measures equality of service and, to the user, the predictability of response time for a request. The measurements indicate that the scheduling system was able to maintain reasonably equal service even when the machine was saturated. As shown in Fig. 8*a*, *b*, and *c*, only one job in 20 experienced a stretch factor as much as twice the mean. The greatest variation occurred in the lightest load (Fig. 8*a*), where



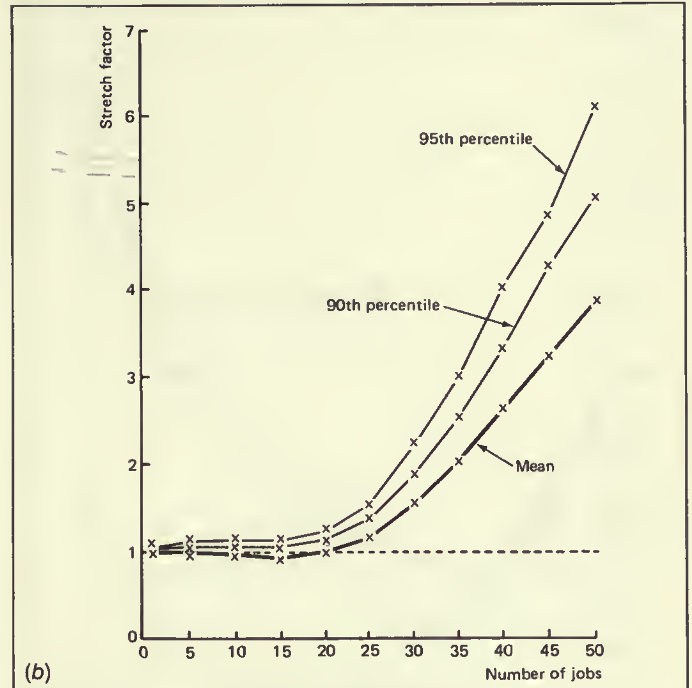
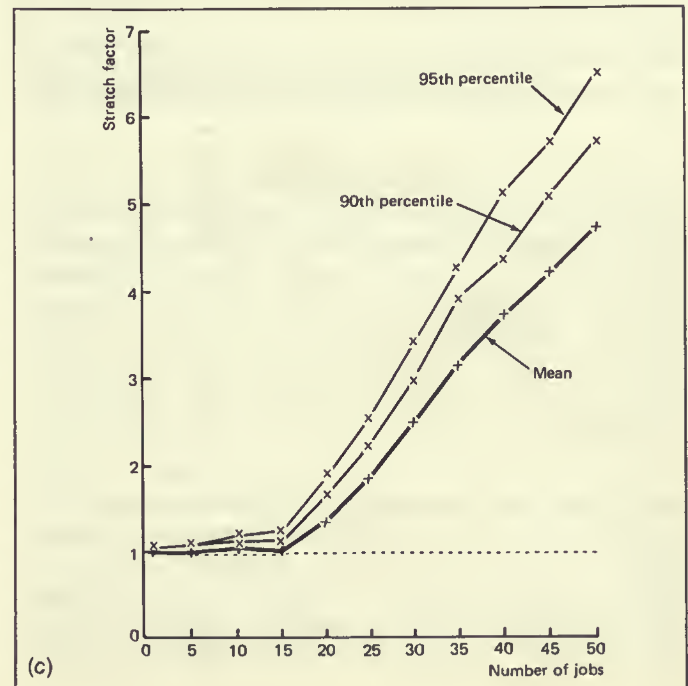


Fig. 8. Variation in normalized response times for job streams of varying computational requirements. (*a*) Low computing load, mean = 1 s compute time per interaction. (*b*) Medium computing load, mean = 5 s compute time per interaction. (c) Heavy computing load, mean = 10 s compute time per interaction.

the effects of scheduling were least dominated by computation. The sharp rise in Fig. 8*b* and *c* indicates the load at which the machine was saturated. The mean stretch factor dips below 1 in Fig. 8*a* and *b* because of statistical variations caused by slight differences in relative Pc and memory speeds.

3.1.4 Memory Contention Although a predictable result of not having implemented Mcache, memory contention has been a problem for high-performance multiprocess application programs on C.mmp. If three Pc's access the same memory port, that port becomes saturated. This limited access resulted in poor performance of multiprocess programs with shared code pages. The solution was to distribute copies of the code pages in different ports to each process. The critical code pages were few, and so the copies did not make excessive demands on memory. Accesses to data were less frequent and sufficiently evenly distributed through the data base not to cause significant contention. Although the code for the operating system is widely shared, its execution is sufficiently asynchronous that memory contention has not been a noticeable problem. Figure 9 illustrates the contention due to shared code pages for the root-finder program.



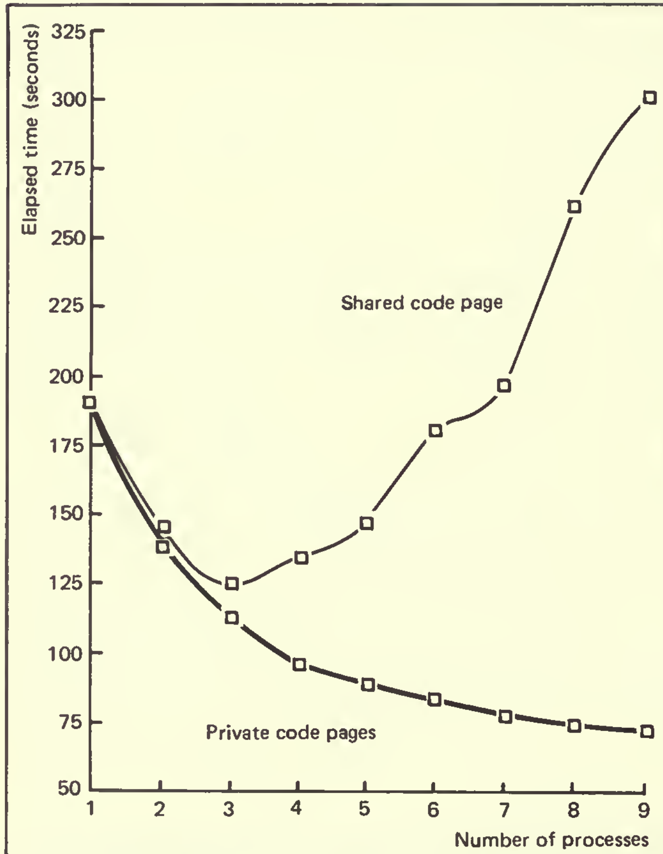


Fig. 9. An example of memory contention with shared code pages.

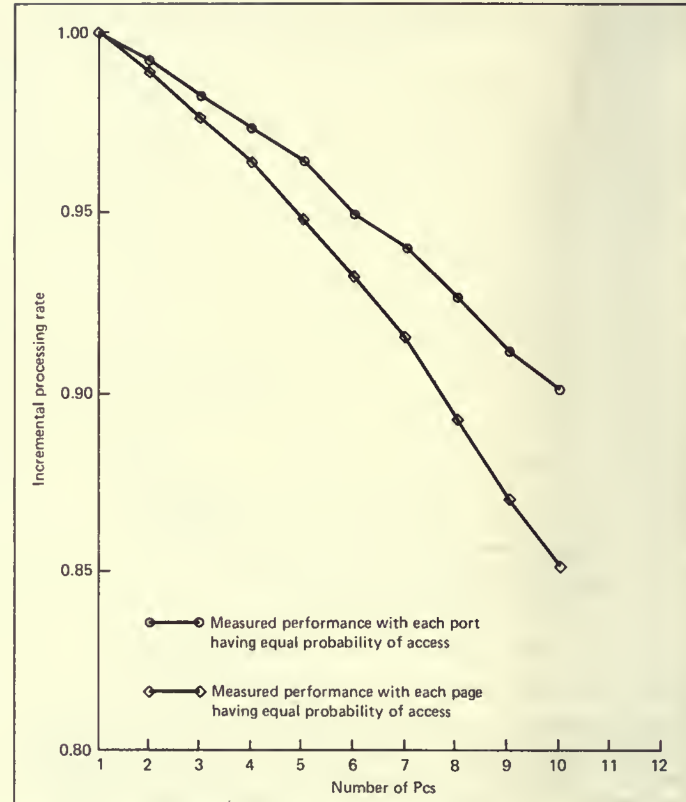


Fig. 10. Processing power for each additional processor.

As illustrated in Fig. 10, detailed stand-alone (without Hydra) measurements of memory contention taken with a synthetic job stream indicate that the incremental value of a Pc is 99 percent for the second Pc and decreases uniformly to 86 percent for the ninth Pc, with a measurement error of about 3 percent [McGehearty, 1980]. The synthetic jobs executed 25 repetitions of a 100-instruction sequence that was chosen as representative of typical instruction mixes for PDP-11's [Marathe, 1977; McGehearty, 1980]. Each processor executed the same instruction sequence, although neither code nor data were shared. After each 25 executions of the 100-instruction sequence, different memory ports for both code and data were independently chosen. The choice of port was either uniform for the 16 ports or weighted by the number of pages available in each port. The selection of different memory ports was repeated 4,096 times, each time including the 25 executions of the synthetic instruction sequence. Since there was no sharing, the results are representative of the general-purpose, time-shared environment envisioned for the machine.

3.2 The Effects of Using Minicomputers

While C.mmp has shown that small processors can be effectively harnessed into a large-scale system, the decision to use minicomputers has not been without problems. Two characteristics of most minicomputers available in 1971 and 1972 have had considerable impact on the project: first, they were not designed for reliability; second, the small word size affects both data representation and addressing. The small address size has proved to be a serious problem.

3.2.1 The Small-Address Problem The small-address problem (SAP) [Wulf and Harbison, 1978] stems directly from the use of minicomputers as the processing elements of C.mmp. The PDP-11, being a 16-bit machine, can address only a 64-Kbyte space. This is much too small for large-scale applications, although

it is often sufficient for individual subsystems of the operating system.

The problem typically appears in addressing an application's data base. Large problems tend to have large data bases, and the 16-bit address allows access only through a 65-Kbyte window. The problem hasn't been as severe for code, because the size of the code has usually been relatively small. In cases where code size was important (e.g., Hydra), subroutine-calling sequences were developed that automatically made the called routine addressable before entry and restored addressability of the caller upon return.

The real cost of SAP is the all-pervasive concern for the addressability of the data during the design and coding of a program. Demand paging was precluded by the 16-bit address and limitations in the relation mechanism. Dmap does not retain sufficient information to identify the address causing a nonexistent memory fault [Levin, Cohen, Corwin, Pollack, and Wulf, 1975]. Therefore the Hydra paging system was forced to require that working-set and addressability changes be written into the program. While the mechanisms the paging system provides are clean and efficient, the necessity of having to explicitly juggle the working set and its addressability results in design and coding burdens. Performance problems, although secondary to programming problems, stem from the frequency of addressing changes. The cost of addressing changes is minimized by a microcoded relocation-register loading function available to user-level programs, and by the fact that the relocation registers are always addressable in the kernel.

In the Hydra kernel, the performance cost of SAP has been measured at 5.5 percent, or an addressing change every 16 instructions [Marathe, 1977]. This is higher than the cost incurred by moderately optimized user programs for two reasons. First is size: Hydra is, by a considerable margin, the largest program executing on C.mmp. It has nearly 50 code pages and from 10 to 100 data pages, depending on load. The order-of-magnitude variation of the data space needed contributes to the frequency of addressing changes by forcing nearly all data structures to be dynamically addressed. Dynamic addressing, in turn, is made more expensive in the kernel by the second reason: the necessity of disabling two relocation registers in the (1,1) addressing space (see Fig. 3). Perhaps if Dmap supported more relocation registers and a smaller page size, the problem (performance, at least) would be somewhat alleviated [Wulf and Harbison, 1978].

Figure 11 illustrates a case study of the effects of the SAP [Wulf and Harbison, 1978]. The task is the Harpy speech understanding system with the 37-word vocabulary. Two versions of the same task are compared: one with dynamic mapping and one with static mapping. In the dynamic mapping version, Harpy checks each data access for addressability; in the static case, the program assumes the data are addressable. Note that a factor of 3 in performance was gained by simplifying the code even though in actuality no addressing changes were ever necessary—the cost was incurred by *checking* for addressability.

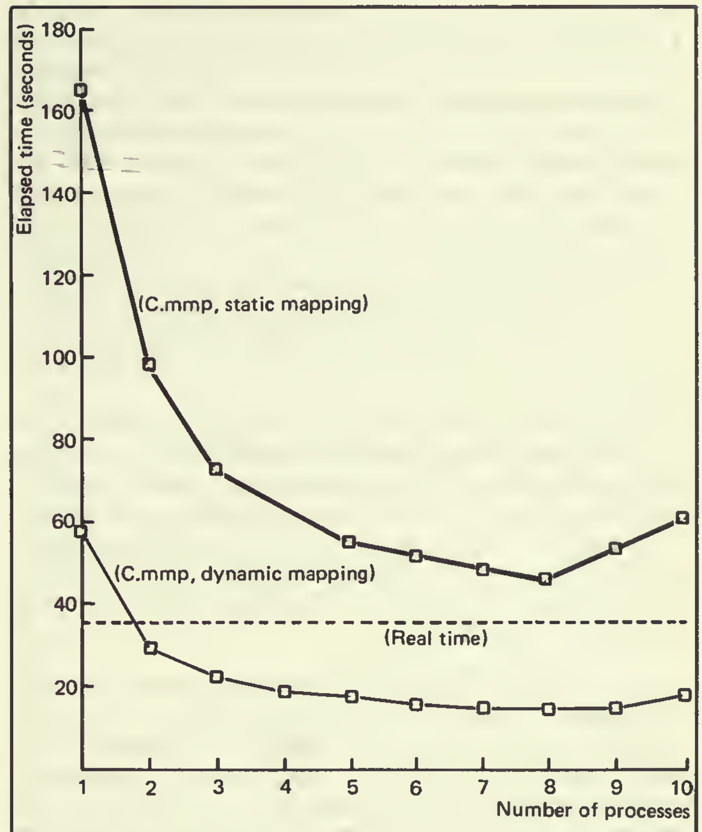


Fig 11. The effect of SAP on Harpy.

Another example is in the coding of the operating system's Command Language (CL). The CL provides a small ALGOL-like programming language (similar to BLISS-11, the implementation language used for Hydra [Wulf, Russell, and Habermann, 1971]), complete with variable declarations and macro facilities. Static data structures were used to implement the CL, with the result that although the code is simple, the size of the symbol table is quite restricted. This has limited the utility of its macro processor in tailoring the user's interactive environment.

3.2.2 The Effect on the Capability Protection System Hydra represents capabilities in 16 bytes, or 8 words. Eleven bytes are required: 8 for the global name and 3 for the rights field. The remaining 5 bytes are allocated to reliability checks and other implementation details. Having to move 8 words per capability is a significant source of overhead in the protected procedure call.

Recent measurements of typical Hydra typecalls, the most frequently executed version of the protected procedure call, indicate that an overhead of about 30 ms is to be expected. Detailed software traces of the calls indicate at least 50 percent of the time was spent in merging the capabilities into the new LNS. While creating the new LNS is the major function of the call, several factors are responsible for its being so expensive: making the capabilities addressable at both source and destination, the locking required for the capability copies, and simply moving the 8-word representation. Since this move cannot be done indivisibly, the locking is required. The typecalls studied were catalog lookups and had fewer than 15 capabilities per LNS. That fraction of the overhead devoted to building the LNS is proportional to LNS length.

The result of the high overhead has been that although the capability-based protection system is highly effective, its efficiency has often limited its use. In particular, protected procedure calls are used on a considerably larger grain than was anticipated. It should be noted that this is not considered a problem inherent in capability systems, but an artifact of implementation with small words and on hardware not specifically intended to support capability addressing.

3.2.3 Indivisible Operations The implementation of locks in a multiprocessor is dependent on having at least one indivisible operation on shared memory. Although not specified in the PDP-11 ISP, the 11/20 and 11/40 both perform indivisible read-modify-write cycles. Smp maintains the indivisibility, making any instruction using this access mode indivisible. Increment, decrement, and shift instructions are used in the construction of the various forms of lock in Hydra. The fact that the bit manipulation instructions are made indivisible automated the synchronization of the bit mask operations so critical in using the IP-bus functions. The richness of the indivisible instruction set has been of great value to C.mmp and should not be underestimated.

3.2.4 Lack of Error Checking The necessity of constructing C.mmp from available minicomputers greatly restricted the possible-fault-tolerant mechanisms that could be incorporated. For example, neither of the two PDP-11 models used, nor the Unibus, has error-checking abilities; one must assume that their results are correct. Experience has shown that this is frequently not the case. Therefore elaborate error checking and correcting of the shared memory and its access path were not justified, because of the possibility of data corruption on the Unibus.

The lack of checking by the hardware forced the burden onto the software, with a resultant penalty in performance. Software checking generally consists of checksums and type and consistency checks. Because data integrity is considered highly important, the error-checking burden falls most heavily on the GST.

3.3 Reliability

In spite of the difficulties, the machine has been reasonably reliable, considering its highly experimental and unique nature. Recent statistics indicate that the total system *mean time to crash* (MTTC) from all causes is, with one exception, fluctuating between 6 and 15 h, averaged on a monthly basis. This is more than enough to be a useful research tool, especially since the average downtime after a crash is only about five minutes and the machine automatically reloads itself (operator intervention is virtually never required). In a research environment, availability has proved to be more important than absolute reliability. Figure 12 illustrates the distribution of crashes during the end of construction and the beginning of an intensive maintenance period. Completion of the machine allowed engineering efforts to be directed to reliability, and the error rate improved accordingly.

3.3.1 Reliability Experience The reliability experience has been quite varied: many failures that were once common are now rare or nonexistent, others are still apparent, and some reappear from time to time. The failure rate has been significantly improved through a program of intensive maintenance, which has been in progress since completion of the basic machine.

Memory parity failures have, with rare exception, been the most common failure mode. Most are transient, but hard errors happen with regularity. Often the memory failure rate has largely determined the MTTC. For example, the sharp peak in Fig. 12 was caused by memory-related errors when the last of the MOS

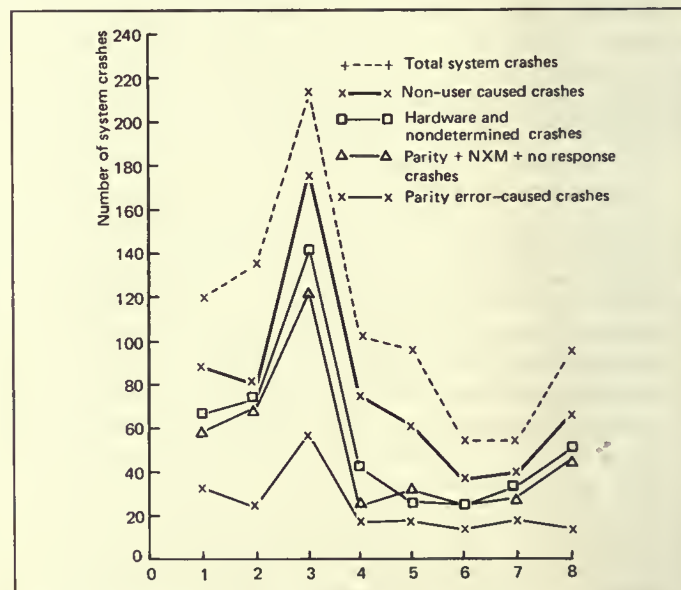


Fig. 12. Distribution of crashes on C.mmp. Eight months data: 1 July 1977 to 8 February 1978.

memory was installed. A methodology for recovering from transient memory failures in the shared memory of the operating system is now being developed, and a marked improvement in reliability is expected from this one recovery effort, since most memory parity failures happen in the operating-system kernel. Memory failures in user-allocated pages present a lesser problem.

Transient failures, while it is always difficult to isolate their source, have been an especially large problem on C.mmp, since there are few, if any, trace points in most data paths. Not including powerful debugging aids in the logical design has continuously hampered development. There was little that could be done for the processors, but aids could have been incorporated into all the CMU-built logic. When this weakness was realized, one tracking register (for the program counter) was added to Dmap; another (for operand addresses) is being developed. A similar weakness became evident in the software: often information about a failure was lost by the operating system, making recording of the conditions for transients unreliable. Robust crash-logging procedures have alleviated this to a great extent.

A transient failure that has eluded solution is the problem of "false NXM's." The processor reports a nonexistent memory (NXM) exception, but upon analysis it is found that the memory is responding and the instruction, registers, and index word(s) are well formed; no exception should have resulted. Because of the lack of checkpoints in the memory data paths, there is insufficient information available to isolate what may be failing.

Another long-standing transient is stack operation problems. This usually appears as misexecution of subroutine call/return instructions or as interrupt entry/exit mistakes. The most common form of the error is one word too many (few) pushed (popped) onto (from) the stack. No cause has ever been isolated, and no method of recovering from this failure has been developed, but, fortunately, it is relatively rare.

A pleasant surprise has been the reliability of Smp. Although it is the most complex component of the multiprocessor hardware, it is now among the most reliable. No doubt the relatively simple design, conservative implementation, and careful construction have paid off. The complexity of Smp is indicated in the chip counts in Table 4, which also includes the expected chip failure rates as calculated by Autofail, a CMU-developed hard-failure model based on the Military Standardization Handbook 217B

Table 4 Chip Complexity and Theoretical Failure Rate for Smp

Logic unit	*Chips*	*Gates*	*Failures per 10^6 h*
Cross-point logic	1,656	29,808	328.9
Priority contention resolution	864	7,344	121.7
Processor interfaces	384	3,552	57.1
Crosspoint enable/disable	544	4,448	77.7
Totals	3,448	45,152	585.4

model [Siewiorek, Kini, Jobbani, and Bellis, 1978; Bellis, 1978]. Figure 13 illustrates the relative simplicity of the four types of PC boards used in Smp.

An early problem with Smp required considerable effort to fix: Certain conditions, characterized by a memory access not completed by the Unibus master, could cause Smp to deadlock on account of the lack of a time-out circuit in the memory port control logic. Any other Pc attempting to access the deadlocked port will block until manually cleared. This situation was often caused by poorly designed I/O controllers that recovered from errors by simply aborting the current access with no regard for proper termination of Unibus or switch protocols.

While the known cases that deadlocked memory ports were isolated and individually remedied, the most important result was an appreciation of the design principle of mutual suspicion [Schroeder, 1972]. The switch should never trust that an operation started will necessarily be completed; it must be prepared to time-out, clear itself, and report a failure condition to the requesting processor.

The IP-bus is as unreliable as the switch is trustworthy. Having no error checking whatsoever, its reliability is so poor that if a cheap and highly effective method of software recovery hadn't been found, the bus would be nearly unusable. The mode of failure is transient loss of interprocessor interrupts and changing interrupt levels—usually from level 7 to level 6. Although no cause has been isolated, a simple system of pending interrupt masks allows an interrupted Pc to determine the validity of the interrupt. The same masks allow automatic repetition of lost interrupts, likely by a different Pc.

Two remaining long-term reliability artifacts of the architecture are:

1 Overrun errors on I/O-device DMA transfers caused by memory port contention. This is a predictable result of not having the planned cache memories and is effectively recovered from in software.

2 Having I/O devices associated with specific processors causes undesirable dependency on that processor. A partial solution has been developed in software to recover from transient failures, but frequent or hard failures force a shutdown for repair or reconfiguration. Fortunately, shutdown is very rare.

3.3.2 Software Recovery Methods within Hydra As the above description of the failures encountered indicates, fault tolerance is the result of a highly cooperative effort between hardware and software. Some failures, such as losing interprocessor interrupts, produce no damage and require so little effort in software recovery that little motivation exists to correct the hardware. Others (deadlocked memory ports) are impossible to recover from with software; much effort has been devoted to eliminating the

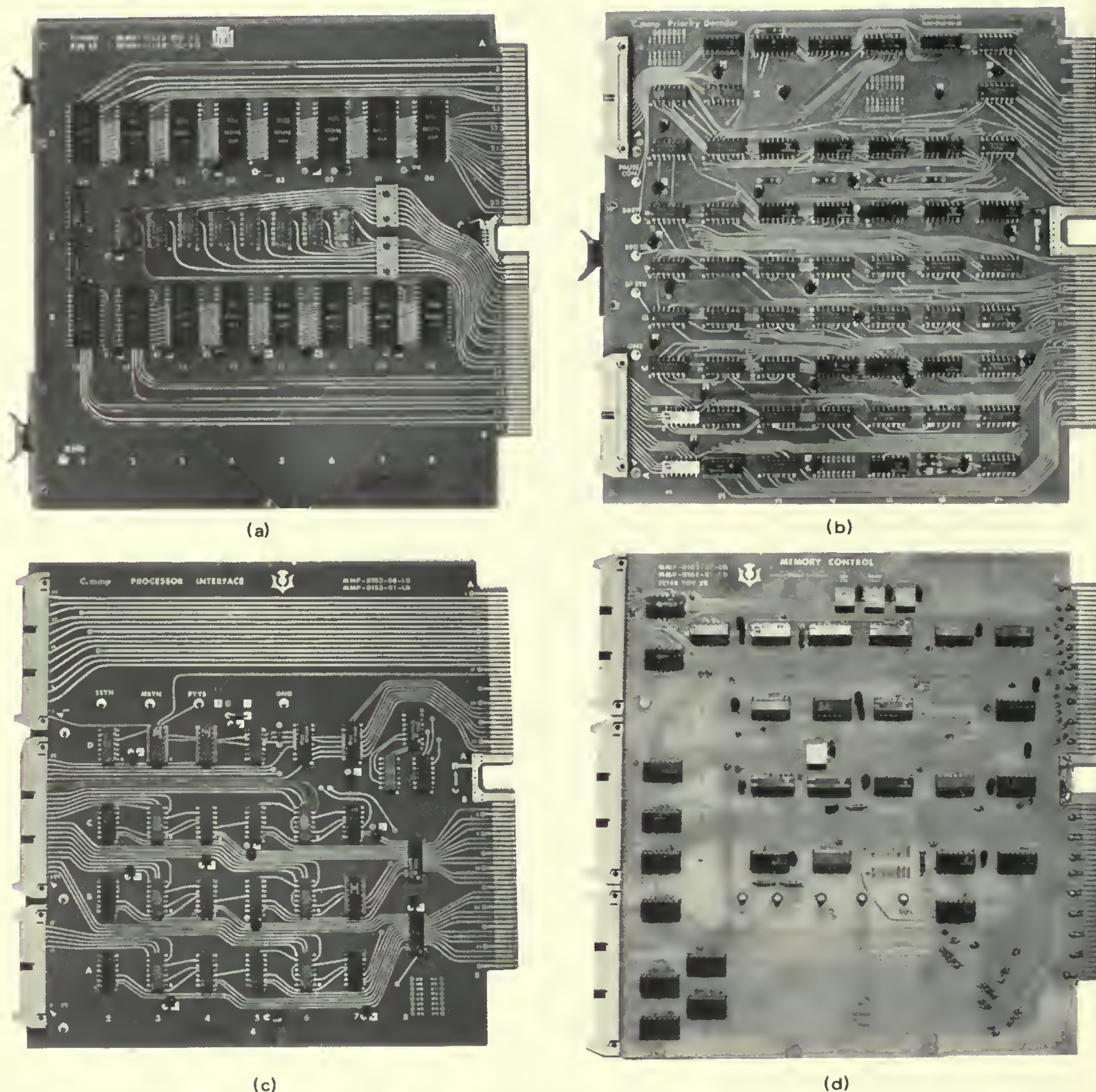


Fig. 13. PC boards used in the crosspoint switch. (a) Crosspoint logic (72 boards). (*b*) Priority contention resolution (16 boards). (c) Pc interface (16 boards). (*d*) Memory interface (16 boards).

sources of failure. The software recovery methods, developed by design and evolution, may be grouped by similarity: methods for recovery from frequent failures that have little probability of nonlocal damage, and methods for treating relatively rare, but serious, failures that may imply systemwide damage.

The first class of failures is typically transient, though frequent, and does not involve shared data structures. Examples are IPI failures, DMA overruns, and memory parity failures in user-allocated pages. Although simple recovery methods of retrying and reporting failure are used to handle these errors, a consistent effort is made to reflect the error report back to a level where there is sufficient information for proper action [Parnas, 1972].

For the second class of errors, those serious enough to imply nonlocal damage, two major techniques have been developed, both of which exploit the parallel environment of C.mmp. In the Hydra kernel, the availability of multiple Pc's is used to create a robust recovery and logging system, and at user level, multiple processes are used in an analogous manner.

Important system elements of Hydra, such as job scheduling

and file systems, are implemented as user-level programs. Their response to errors is critical to system reliability, and several multiprocess techniques are used. The processes may be multiple incarnations of the subsystem's server processes, or they may be free-running *daemon* processes created specifically to play a watchdog role in ensuring the correct and reliable operation of the subsystem. The multiple-incarnations approach accepts the loss of a server and the processes dependent upon it as a method of limiting damage and also tends to improve response. The daemon approach is specifically creating redundancy for reliability.

Within the kernel, serious errors are handled by a formal mechanism, *the suspect/monitor* model, which causes the whole system to pause so that a known state is reached before a sequence of error logging and analysis is performed. This procedure allows a wide range of options, from continuing execution, possibly with configuration changes, to reloading (again, possibly reconfigured). Developed in response to the low reliability of the developing hardware and software, suspect/monitor was retrofitted to the existing software.

Invocation of the suspect monitor sequence may occur in two ways: First, a Pc may detect an error condition either by hardware trap or software check. It then becomes the *suspect*, and a *monitor* Pc is randomly chosen from the remaining processors. Second, a Pc executing the *watchdog* routine detects that some other processor has apparently not been executing. The watchdog processor becomes the monitor and declares the apparently nonexecuting Pc to be the suspect. The watchdog routine is executed by all processors as part of several frequently used interrupt service routines and sets a bit (corresponding to the executing processor) in a mask maintained by the watchdog. Periodically this mask is compared with a mask of Pc's known to have completed initialization (upmask) and then cleared. Any processors in the upmask but not in the watchdog mask are declared suspects.

Once the monitor is chosen, it and the suspect achieve synchronization by means of a shared-state variable. Each advances the variable to the next state upon entry. Both examine the state, and if it is not in the synchronized state, each waits for the other to advance it to that state. The monitor times all waits for the suspect to reach a desired state, and if synchronization is not achieved quickly, the monitor attempts to force the suspect Pc to execute the recovery code with a sequence of IP-bus operations. Continued failure to synchronize causes the monitor to abort the sequence and force a reload. Multiple suspects are processed one at a time by the same monitor.

The suspect's sequence is: record all Pc state at the time of failure, including which pages were addressable; copy its local memory; execute a short-diagnostic; and, assuming correct execution of the diagnostic, attempt analysis of the failure. Completion of these actions is communicated to the monitor via the state variable. Because of the sensitive nature of the suspect's execution, several coding restrictions were employed in its implementation. For reliability, no stack operations are performed, the Pc state–logging code is straight-line, and a flag is set upon entry to the suspect routine to force an immediate halt upon repeated entry for any reason. Halting causes a monitor time-out, forcing a reload and preventing the previously logged data from being overwritten.

Once synchronized, the monitor follows the suspect through its sequence and, after successful completion, has the following options:

- Continue with no changes.
- Halt the suspect and continue.
- "Quiesce" the suspect and continue.
- Reload.
- Reload and delete suspect from configuration.
- Reload and quiesce the suspect.

Quiescing a processor allows it to service I/O device interrupts but not to execute any other functions (notably user programs). This way, the duty cycle is kept low, and it is hoped, so is the probability of a failure. This mode is required to keep processors with critical I/O devices in the configuration. Since most data structures lack the redundancy and associated verification routines to guarantee repair of damage, all paths through suspect/monitor currently lead to one of the system reload options.

The analysis that the suspect may perform is highly failure-dependent. Because of the problems of installing any recovery scheme in an existing large program, the problems of analysis are only beginning to be examined. Recovery from memory parity failures during kernel execution is being considered as the first candidate for analytical recovery. These parity failures are considered serious enough to invoke suspect/monitor because of the requirement to maintain the integrity of the GST. Also, a page may hold segments of many objects, and so a failure may imply future trouble if not caught promptly. For parity failures, the analysis must ascertain three things: whether the failure is repeatable, whether it happened during interrupt service, and whether any critical data structures were locked. If any of these is true, recovery is not possible. There is no way to report the failure to the process while servicing an interrupt. If locked, a data structure may be in an inconsistent state. In these cases, the suspect notifies the monitor to reload the system. Otherwise, the failure has occurred during a kernel call and may be aborted with a parity failure report. The caller may then decide whether to retry the call. No claim is made that this particular method is optimal; it is intended to illustrate the role of analysis in the suspect/monitor. However, it does promise a high probability of recovering from the majority of parity failures with an acceptably small risk of undetected damage.

The auto-restart mechanism is responsible for reloading the system and is invoked by the suspect/monitor mechanism. Three basic steps are involved: adjusting the configuration masks for any deleted or quiesced processors, constructing a free memory list (deleting pages that have been marked errant), and loading a fresh copy of the kernel from disk. The new system is entered and initialization begins. This sequence is normally accomplished without human intervention and is so reliable that C.mmp runs without an operator.

The last mechanism associated with failure recovery is the automatic diagnostic driver, which initiates and monitors the deleted processors' execution of a diagnostic. The driver maintains a history of the failures found by each processor as well as the processor's successful executions of the diagnostic. The histories may be printed on command and are also accessible from Hydra. If a processor is able to successfully run the diagnostic for a period of time determined by its failure history over the previous few days, the driver automatically returns it to the system. Automatic return is accomplished by executing the standard per-processor initialization and does not require pausing or reloading the system.

4. Conclusion

The successful implementation of systems such as Harpy, ZOG, several language compilers, several file and directory systems, ARPANET support, and measurement tools such as the script driver has shown that C.mmp and Hydra provide a useful, general-purpose computing environment on a multiprocessor. The symmetric design of C.mmp has proved to be valuable in error-recovery techniques and in simplifying process scheduling. Also, the kernel approach to operating-system design, the protection system, and the mechanisms for data abstraction have effectively allowed construction of much of the operating system as user-level programs.

The problems, such as reliability, memory contention, and the small-address problem, have been effectively managed, if not solved entirely. These problems were challenging and the reliability problems, especially, motivated a profitable research effort.

References

Almes and Robertson [1978]; Bellis [1978]; Bhandarkar [1972]; Cohen and Jefferson [1975]; DEC [1972]; Dijkstra [1968a]; Fuller, Almes, Broadley, Forgy, Karlton, Lesser, and Teter [1976]; Fuller and Harbison [1978]; Jain [1978]; Levin, Cohen, Corwin, Pollack, and Wulf [1975]; Lowerre [1976]; Marathe [1977]; McGehearty [1980]; Newcomer, Cohen, Jefferson, Lane, Levin, Pollack, and Wulf [1976]; Oleinick [1979]; Oleinick and Fuller [1978]; Parnas [1972]; Robertson and Ramakrishna [1977]; Rubin, Guggenheim, and Bihary [1978]; Schroeder [1972]; Siewiroek, Kini, Joobbani, and Bellis [1978]; Strecker [1971]; Swan [1976]; Wulf and Bell [1972]; Wulf, Cohen, Corwin, Jones, Levin, Pierson, and Pollack [1974]; Wulf and Harbison [1978]; Wulf, Levin, and Harbison [1980]; Wulf, Levin, and Pierson [1975]; Wulf, Russell, and Habermann [1971].