Reflections in a pool of processors— An experience report on C.mmp/Hydra*

by WILLIAM A. WULF and SAMUEL P. HARBISON
Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

This paper is a frankly subjective reflection upon the successes and failures in a large research project—the construction of a multiprocessor computer, C.mmp, and its operating system, Hydra—by those most intimately involved in its design, construction, and use.

C.mmp and Hydra have now reached a sufficient level of maturity to establish themselves as useful and reliable computing resources at Carnegie-Mellon University. The user community has grown from primarily operating system implementors to include researchers in other operating systems and multiprocessors and casual or curious users interested in using the unique features of the system (e.g., the Algol 68 language, whose first implementation at CMU was on C.mmp.).

Some of the scientific results we originally hoped for have been published and are listed in the bibliography at the end of the paper. Other results will be published in the future as we observe the system under varied loads and over longer periods of time. In addition to these factual results, however, we have learned a number of things of a more subjective nature—things that we did right and, perhaps more importantly, things that we did wrong. We believe that many of these lessons are not unique to our project, and their presentation here will be valuable to the larger computer science community.

For those people unfamiliar with C.mmp and Hydra, we shall provide a brief overview of multiprocessor research at CMU, and some details about C.mmp, Hydra, and the goals we originally set for the research project. This information should serve as a general background against which our evaluation of the project can be cast. The interested reader will find more details in the bibliography.

Multiprocessor research at CMU

In late 1971 we at CMU decided to embark on a research program to explore multicomputer structures—especially those structures in which the several computers share a common address space. At the time it appeared to us that the economics of LSI technology would make multi-mini or multi-micro structures the architecture of choice for many medium- to large-scale applications. In addition to the economic arguments, there appeared to be many other advantages to such structures, including high availability, expansability, and so on.

Despite the fact that a number of multiprocessor computers had been built prior to 1971, relatively little of a scientific nature was known about them. Our goal was to explore a number of alternative multiprocessor designs, examining both the hardware and software issues, and to report on these explorations. To that end we undertook the design and construction of two multiprocessor systems, C.mmp and Cm*, and their associated software.

C.mmp, the subject of this paper, is a relatively straightforward multiprocessor. Begun in 1972, it connects 16 processors to a large shared memory (up to 32 megabytes) through a central crosspoint switch. The access time from any processor to any word of memory is identical. Cm*, started in 1975, replaces the crosspoint switch with a distributed, bus-oriented interconnection scheme between processor-memory pairs. In contrast to C.mmp, the access time from a Cm* processor to a word of memory can vary by an order of magnitude depending upon the particular processor and memory module involved. These two machines have quite different implications on the software which runs on them; between them we are able to explore many of the interesting issues of distributed processing.

C.mmp

C.mmp is a multiprocessor composed of 16 PDP-11's, 16 independent memory banks, a crosspoint switch which permits any processor to access any memory, and a typical complement of I/O equipment. A path through the switch is independently established for each memory request and up to 16 paths may exist simultaneously. An independent bus, the IP-bus, carries control signals from one processor to another; no data is carried by this bus. Collectively the 16 processors execute about 6 million instructions per second;

* The research described here was supported by the Defense Advanced Research Projects Agency (Contract: F44620-73-C-0074, monitored by the Air Force Office of Scientific Research). The views expressed are those of the authors.

the total memory bandwidth is about 500 million bits per second. In short, despite the fact that it is built from minicomputers, C.mmp is a large-scale machine.

The current configuration of C.mmp includes 5 PDP-11/20 processors (5 usec/instruction), 11 PDP-11/40 processors (2.5 usec/instruction), and 3 megabytes of shared memory (650 nsec core and 300 nsec semiconductor). All of the 11/40 processors have been modified to include writable microstores; thus we are able to tailor their instruction sets to specific applications. The cost of this configuration is roughly $600,000, of which $300,000 is the cost of processors, $200,000 is memory, and $100,000 is the switch, IP-bus and other special equipment. Of course, there is an additional cost associated with I/O devices.

Hydra

Hydra is the "kernel" of the operating system for C.mmp; it is not intended to provide most of the familiar features of an operating system (e.g., it does not provide files, a command language, or even a scheduler). Rather, Hydra provides an environment in which it is (intended to be) easy to write user-level programs that supply these familiar facilities. Hydra was designed in this kernal fashion in order to permit (and encourage) experimentation with features and policies appropriate to multiprocessors.

Hydra, which was a research project in its own right, uses a capability-based protection structure, a scheme in which only the possession of the appropriate kind of reference to an object (e.g., a file) grants access to that object. In order to allow user-level definition of operating system facilities, Hydra extends the basic capability scheme with the ability to define new types of objects and (protected) operations on these object types. Thus it is possible for a user to define new types of files, processes, message buffers, or whatever. These newly defined types share an equal status with those that already exist—which is another way of saying that Hydra attempts to preempt as few decisions as possible, thus allowing the users to tailor the system to their needs.

Software already built on top of Hydra in this manner includes file systems, directory systems, schedulers, and language processors (for Algol 68, L*, and a flexible command language).

Project goals

Two general goals influenced both the hardware and the software design from the outset. The C.mmp/Hydra system was envisioned as both *symmetric* and *general purpose*. By *symmetric* we mean that replicated components, such as processors, are treated as an anonymous pool; no one of them is special in any sense. By *general purpose* we simply mean that we did not intend to cater to *only* those programs which need a multiprocessor; the multiprocessor character of the machine is used to improve throughput across a set of independent jobs as well as to multiprocess single jobs. Both the hardware and software were designed with these goals in mind.

The *symmetry* goal is manifest in a number of ways. At the hardware level, for example, an interprocessor interrupt mechanism was designed so that every processor could interrupt every other processor (including itself) with equal ease. At the software level there is no "master-slave" relation among the processors—any processor may execute any part of the operating system at any time (subject, of course, to mutual exclusion in accessing shared data structures). At the user level, a job may execute on any processor, and indeed may switch from one processor to another many times during its execution.

The impact of the *general purpose* assumption is more subtle; it implies that we have to provide a broader range of software than would be expected if our focus had been more narrow. It also implies that optimizations to a specialized problem domain should not be made in the operating system. Some of the specific effects of this goal will be found later in the evaluations.

Performance evaluation tools

Many of our evaluations of C.mmp are based on data obtained from a number of tools designed to measure system performance. Although not one of our greatest successes, we think these tools are important enough to present here. We have three measurement tools: a script driver, a hardware monitor, and a kernel tracer.

The Script Driver is a program which can place a measured load on the system by simulating a number of users at terminals performing various tasks. This known load can make the interpretation of performance measurements much easier.

The Hardware Monitor is a device built at CMU which can monitor in real time the signals on a PDP-11's bus. The Monitor is very useful in measuring the activity of a single C.mmp processor, and for recording the activity of small portions of the operating system. It is less effective in measuring total system performance.

The Kernel Tracer, the most commonly used tool, is built into the Hydra kernel. It allows selected operating system events (e.g., blocking on semaphores, context swaps) to be recorded while applications are running. The accumulated data can be processed off-line to give a detailed record of what was happening on each processor. Naturally, the use of the tracer slows down the entire system, but this obvious point doesn't really seem to matter in practice.

The importance of these tools should not be underestimated. In any system as complex as an operating system, design decisions are often based on intuitive assumptions of performance tradeoffs. Without accurate measurements, these design assumptions cannot be verified. Certainly we found that some of our assumptions were wrong, causing us to redesign several parts of Hydra.

Format of the paper

The body of this paper is a highly edited report of a meeting called specifically to evaluate the C.mmp/Hydra

project. The attendees were representatives of the various groups involved in the design, implementation, and use of C.mmp and Hydra; hardware designers, operating system implementors, those doing performance evaluation, and four major users. In all, sixteen persons attended, the maximum number we felt could interact productively.

The purpose of the meeting was to solicit the opinions of the participants concerning the nature of our successes and failures. We had also solicited written opinions from a wider group—in fact, just about everyone who has had anything to do with C.mmp and Hydra. The participants knew, of course, that the results would be reported in this paper.

The meeting and written responses produced over a hundred distinct comments. To organize these in a coherent fashion we asked the participants to decide upon our five greatest successes and five greatest failures. With some exceptions the comments have been organized under these headings; the participants' comments have been indented to separate them from background information and summary comments.

Any paper that sets out to reflect upon the successes and failures of a research project is potentially self-serving. We were extremely conscious of that danger and have attempted, through the format of the meeting and the editing of its transcript, to construct the paper in a manner which minimizes this effect. Either our initial fear of being self-serving was groundless, or the format chosen worked extremely well. We shall let the readers judge for themselves, but we feel that the result has been a reasonably objective, well-balanced view of the C.mmp/Hydra project.

OUR GREATEST SUCCESSES AND FAILURES

We shall begin this report with what, in fact, happened last at the meeting—a listing of our most notable accomplishments and mistakes. This list was created after all opinions had been expressed, thus the participants had the opportunity to hear the opinions of the others before deciding upon the content of the list. To keep the discussion crisp we arbitrarily chose to limit each list to five items. Surprisingly (to the editors at least), despite the differing interests of the participants there was essentially complete agreement on the items to be included on each list.

Our notable accomplishments:

- We constructed a cost-effective, symmetric multiprocessor.
- We provided, in Hydra, a capability-based protection system which allows the construction of operating system facilities as normal user programs.
- We were able to distribute the Hydra kernel symmetrically over all processors.
- We provided successful mechanisms for the detection of, and recovery from, software and hardware errors.
- We used an effective methodology for constructing the Hydra kernel.

Our notable disappointments:

- The hardware is less reliable than we would like.
- The small address of the PDP-11 has a large negative impact on program structure and performance.
- We are unable to partition C.mmp into disjoint systems.
- We did not put enough human-engineering into the software interface to the user.
- We did not give enough attention to project management.

Neither our successes nor failures are, of course, unqualified, and the story behind each is littered with smaller successes and mistakes. Moreover, there are dependencies between the things that went well and those that didn't; the fact that we have a running 16-processor system must be tempered, for example, by a poorer-than-expected reliability record. The reliability record, on the other hand, led us to greater concern for software structures that detect and survive hardware malfunction—and we count those structures among our most important accomplishments. For all these reasons, while we have used the success/failure list to organize the paper, one should not expect all the points listed under a "success" to be positive in nature. On the contrary, we believe it important to expose the contributing events, both positive and negative, as well as the major points listed here.

With that introduction then, here is the report of the meeting.

THE SUCCESSES

A cost-effective multiprocessor

C.mmp's design goals included speed, simplicity, and the use of as many commercially-available components as possible. Because C.mmp is a unique computer some critical parts had to be designed and built especially for the project. While this was a burden, it did give us maximum freedom in the design of these critical components, including the crosspoint switch, the IP-bus, and the processor modifications for memory relocation. These were all built by the CMU Computer Science Department Engineering Laboratory.

- The basic design goals have been justified by experience, with speed having been the least important emphasis.
- CMU-built hardware is not a large proportion of the total system cost.
- The crosspoint switch is very reliable, and fast enough.
- The use of immediately available components was a major factor in getting C.mmp built as fast as we did, but it limited us in taking advantage of technology which developed in succeeding years.

We were especially happy about the evaluation of the crosspoint switch, which many people thought would be C.mmp's Achilles' heel. In retrospect we think we were too concerned about raw speed in the design of the switch and

memory; as it turns out, most applications are sped up by decomposing their algorithms to use the multiprocessor structure, not by executing on a processor with short memory access times.

The comments at the meeting did reflect some specific complaints about the hardware, several of which we later decided were significant enough to be listed as some of our major disappointments. Many of these stemmed from our choice of a processor for C.mmp. In 1971, only the PDP-11/20 minicomputer met our requirements. In 1974 we decided to take advantage of technology advances and use the new, faster PDP-11/40 processors to complete C.mmp. One feature of the PDP-11 architecture which might be expected to impact the goal of symmetry for C.mmp is the close association of an I/O device with exactly one processor.

The PDP-11 processors required more modifications than we expected to ensure the security of the operating system.

The PDP-11's 16-bit address is too small for many interesting applications.

Having to supporting two PDP-11 models complicated the development of the processor modifications and the operating system. It would have been better to have had a single processor model, regardless of its speed.

Having I/O devices bound to particular processors made it difficult to move a device from a malfunctioning processor to a good one, but device utilization was not otherwise sacrificed.

Perhaps more than anything else, our experience with the PDP-11 has given us a much clearer idea about what features are really important in choosing a processor, and which are not. Our consensus is that speed is not very important, for reasons already cited in conjunction with the crosspoint switch. Reliability is very important, but we found that much can be done in software to increase the overall system reliability, as long as the hardware has some basic error-detection mechanisms. (Our own approach to this is described later.) The address size is important because if it is too small for the expected applications, the ensuing problems cannot be completely overcome by software. The PDP-11 I/O architecture is an example of a feature that turned out to be unimportant because it could be completely hidden from users by software.

At a higher level, users of C.mmp seemed satisfied with the overall system performance.

Our ability to support multiprocess algorithms is well established by the performance of the many applications on C.mmp.

We have successfully supported user processes that require real-time response, although this was not one of our major goals.

At the end of the paper we will give some performance figures for an application which runs on several CMU computers, including C.mmp.

Most often cited criticisms of the system were:

Interaction with operating system facilities, in or out of the kernel, is accompanied by a high overhead.

The most serious obstacle to rapid execution of large systems is the limitation imposed on programming by the small PDP-11 address.

Memory contention significantly degrades performance when many processes are accessing the same memory page. This is usually caused by the processes sharing the same code pages.

Memory contention is very serious when using high-performance I/O devices which depend on rapid access to memory during transfers.

The performance bottlenecks are due to a combination of avoidable and unavoidable factors. We were initially distressed at the high operating system overhead (it takes about 500 microseconds to enter and exit the kernel), but we attribute most of it to a lack of experience with the fairly complex features we wished to implement. We are confident that the overhead is not an inevitable result of our protection mechanisms, nor is it due to the hardware design.

Memory contention, caused by several processors trying to access the same memory simultaneously, was a performance concern from the outset of the project. Our simulation studies indicated that its effect would be minimal, but in practice several circumstances conspired to make the problem significant. First, typical large multiprocess applications tend to share the same code among all processes, and this greatly increases the probability of accesses to the same memory. Second, the installation of per-processor caches, which were to handle this code-reference problem, has been delayed due to various resource shortages. Finally, we found that devices such as our disks and drums could not tolerate the long memory access times characteristic of periods of high contention. A software solution to this problem had to be implemented.

The small address problem is serious for large applications which cannot fit within the 64K address space on the PDP-11. Although we could not have avoided this problem, we were guilty of underestimating its significance for the applications which were to run on C.mmp. The problem is considered in more detail later in this paper.

Protected subsystems

In Hydra, the construction of operating system facilities outside the kernel is centered around an abstraction called a *protected subsystem*. A subsystem is, in its basic form, a new object type combined with a set of procedures which operate on objects of that type.

Our experience derives from over twenty working subsystems implementing schedulers (*Policy Modules* in Hydra terminology), files, directories, an I/O device allocator, and a host of other traditional operating system facilities. As

software development continued by diverse users, we were curious to see whether all the required software could be built within the subsystem abstraction, whether such development could be done easily and quickly, and whether the resulting facilities could be easily merged into the user environment.

The protected subsystems abstraction is very powerful in designing operating system software in a capability environment.

It is easy to design subsystems which are easy to use and which are protected from any interference from software outside the subsystem.

The subsystem structure makes it easy to provide several coexisting and competing facilities.

The subsystem structure is useful for isolating facilities under development or being debugged.

New subsystems are easily incorporated into the standard system.

We think the subsystem concept in Hydra is as useful as the closely-related notion of extended data types has been in the field of programming languages. Part of the original motivation for the subsystem concept was our desire to allow alternate solutions to problems which we could not foresee in a multiprocessor environment. However, we found that subsystems are also very useful in debugging versions of "standard" systems without interfering with users.

Many people at the meeting were critical of the failure to follow up the subsystem design with the software tools which would encourage building subsystems in this new environment.

Subsystem construction still suffers from being ad hoc, there being inadequate software support for managing the programs, data structures, and documentation which comprise the subsystem.

The development of system software (subsystems) by many different people makes it more difficult to impose any standardization.

Subsystems are less likely to be successful when they attempt to implement traditional (non-capability) systems in traditional ways.

These problems are the result of our not giving the user environment outside the kernel as much attention as we gave the Hydra kernel itself. We consider it one of our worst mistakes and will discuss it more later in the paper.

Scheduling is an example of a traditional operating system function which, in Hydra, is partially implemented outside the kernel by a subsystem called the Policy Module (PM). We thought that providing scheduling policy outside the kernel would allow us to experiment with different specialized strategies for scheduling cooperating processes.

The first Policy Module is a distinguished subsystem for several reasons. First, it was one of the first subsystems built outside the kernel and exhibits many of the mistakes of any first attempt. Second, it is a particularly nice example of our ability to build operating system facilities outside the kernel. Finally, it interacts very closely with the kernel, so the efficiency of the kernel interface is emphasized.

The first Policy Module was operational from 1974 through May, 1977. Our basic evaluation at the meeting was that

The first Policy Module adequately demonstrated that traditional policy decisions could be made outside the kernel.

In spite of this, many people noted flaws in the implementation which were glossed over in our rush to see if the PM would work.

Insufficient attention was paid to reliability and throughput in the Policy Module.

The PM-kernel interface turned out to be more complex than we had anticipated.

We included things in the kernel facilities which logically belonged outside; this acted to complicate the kernel interface. [*For efficiency reasons, we implemented in the kernel some facilities which should have been outside according to our philosophy.*

Hence,

The construction of Policy Modules was not as easy as we had imagined before we actually tried it.

Because we expected a PM to incorporate specific knowledge about the processes it was scheduling, we anticipated having many PM's simultaneously scheduling different sets of processes. Indeed, having several PM's run at the same time was no problem, but again the performance left something to be desired.

To support multiple Policy Modules, more facilities are needed in the kernel to ensure a fair allocation of processor and memory resources to each Policy Module.

We began to build a second version of the Policy Module almost as soon as the deficiencies in the first were recognized. This design proceeded in parallel with performance improvements to the first PM, and in fact we were running both PM's simultaneously for a short time.

The distributed operating system

Hydra was designed with no master-slave relationship among processors. With the exception of the lowest level of I/O device support, all system tasks may run on any and all processors. An immediate result of this is that we expected

a high degree of parallelism in Hydra and the corresponding need for effective synchronization methods.

There are two notable aspects to our approach to synchronization. First, we decided to synchronize on data rather than code. Every data structure which can be accessed by more than one processor is provided with a lock or semaphore which is used to ensure mutual exclusion.

Second, we provided a range of synchronization primitives, from very fast "locks" to much slower "semaphores." The tradeoff here is the overhead needed to P or V the lock or semaphore against the resources which will be tied up by a process waiting to pass the lock or semaphore. Small data structures which are locked for short periods of time (order 300 microseconds) use locks, which involve a very small overhead (approximately four instructions) when the process does not block. Large data structures, or data structures whose processing may be interrupted for long periods of time (as when waiting for I/O) use semaphores, which tie up fewer resources when blocking is necessary.

The simple, symmetric hardware has permitted a much simpler operating system design.

Hydra hides the processor-device correspondence so well that most of Hydra, and all the software at the user level, is unaware of the actual location of I/O devices.

The symmetric distribution of the operating system has been an unqualified success. We are able to achieve a high degree of parallelism within Hydra, and the system is insensitive about the number of processors available.

The use of asynchronous processes ("demons") to implement system functions resulted in simpler designs and improved performance.

In providing synchronization within the kernel, we believe we profited by locking data structures rather than code.

Our decision to provide several types of synchronization mechanisms gave us much design flexibility.

The natural synchronization primitives and our conscious and constant commitment to a high degree of parallelism has resulted in our encountering few software bugs caused by inadequate synchronization.

We have found that the use of demons to absorb much of the system work load outside the normal computational stream has simplified much of Hydra's design. We might not have used this technique if we did not have so much confidence in our synchronization techniques and our ability to achieve a high degree of parallelism.

Coverage of hardware and software errors

There are times when clouds do have silver linings. From the earliest days of the project we had to contend with unreliable hardware and our own software mistakes; moreover, we could not afford a 24 hour/day operator to reload the system after each crash. Thus we were forced to consider the general problems of software detection and recovery from errors—whether they be hardware or software induced.

When an error is detected by Hydra, we try to answer a number of questions. What was the exact error? Can we tell if it is due to a hardware or software malfunction? If hardware, is the problem repeatable or transient? Have any critical data structures been damaged? If so, can the damage be repaired? Can we eliminate a piece of malfunctioning (or just suspicious) hardware and still run? In all cases, our aim is to keep the system running with as much functionality as possible.

Our probability of detecting an error soon after it has occurred is increased by building error-detection mechanisms into the hardware and software. The CMU-built memory relocation units implement parity checking on every memory byte and on the address bus through the crosspoint switch. Software modules employ redundant representation and other techniques to try to limit the propagation of errors not detected by the hardware.

Recovery mechanisms invoked by the detection of any error employ a "suspect-monitor" paradigm to ensure that a failure in the recovery processor may be detected cleanly. Two processors are always involved; one, the *suspect,* attempts to record the system state at the time of the error; the other, the *monitor,* watches the suspect and assumes control if the suspect is unable to finish. The suspect is always the processor on which the error occurred. The monitor is selected at random from all other processors. There are a number of steps which can be taken during a recovery action depending on the type of error, including removing processors or memories from the system and producing extensive crash dumps for later off-line analysis.

The fault tolerance built into some kernel modules resulted in making them among the most reliable in the system—more reliable than other modules coded by the same programmer without using such techniques.

The software facilities for detecting software and hardware errors and restarting the system automatically have been a big success.

Similar facilities in user software are beginning to be developed and show much promise in improving overall system reliability.

Even though we are proud of our current error-handling mechanisms, we know that system needs more work in this area, particularly in the area of supplying policies to determine which mechanisms should be invoked for different types of errors. While it is true that we can recover from virtually any error by initiating an automatic reloading of the operating system, this is a drastic action we would like to use only in the case of truly catastrophic errors. Unfortunately, the difficulty in pinpointing the exact location of some hardware errors and the difficulty in verifying the consistency of the complex capability data structure has resulted in our classifying almost all errors as "cata-

strophic" in this sense. We are in the midst of redesigning both hardware and software to correct these deficiencies.

Software development methodology

Our initial goals for the Hydra implementation did not explicitly include the notion of exploring a software engineering methodology. Nevertheless, we used a method based on Parnas' "modular decomposition"* and it worked quite well; indeed many of us believe that without it the project would not have succeeded.

The methodology used caused us to divide the units of work (programming tasks) along the lines of the major data structures in the system. A module (and hence a programmer) was responsible for the representation of, and all operations on, a data structure. No one other than the responsible programmer had access to knowledge concerning the implementation details.

Because methodology *per se* was not our major goal we were not fanatical about enforcing the methodology, and were often less precise about the specifications than we might have been. Both the positive and negative aspects of this informal approach are reflected in the following remarks:

We believe that it is a measure of the success of the modular implementation of the kernel that one full-time programmer can maintain this program which comprises 2000 (listing) pages of source code.

The independent implementation of the modules in Hydra resulted in a lack of any uniform coding style and in some duplicated effort in interfacing to the underlying hardware. The effect was not very serious since all the implementors were highly talented, exhibiting differences in style rather than quality.

Because modules were implemented independently, no one initially had a detailed knowledge of the entire system. This made debugging more difficult and resulted in a difficult transition when Hydra began to be maintained by a single programmer who was not part of the original implementation team.

Coding of the kernel began quickly after the initial design. Some think too quickly.

Loose management coupled with the modularization technique worked well except in promoting a standardization of coding styles.

Information hiding as a modularization technique resulted in coding situations in which information necessary to make a decision was not available.

As Hydra developed and was modified, the original, clean modularization began to break down as new features were added and performance bottlenecks removed.

* Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *CACM*, 15, 12, pp. 1053-1058, 1972.

We still think the modular decomposition methodology is extremely good for structuring large systems. In our experience, breakdown of the modular structure occurs mainly when programmers in the midst of debugging adapt "quick and dirty" solutions which do not preserve modularity.

All but a very small part of Hydra is written in a high-level implementation language, Bliss-11. There seems to be no question that it was possible, indeed advantageous, to write the kernel in Bliss, but there were problems. The Bliss-11 compiler was developed only shortly before the kernel was begun and was an independent research project (investigating compiler optimization techniques). There was some initial friction between the two groups, but both appear to have benefited in the long run.

The Bliss-11 compiler was designed to compile a slightly modified version of the Bliss-10 language into very compact PDP-11 code. This it does.

The implementors of the Hydra kernel were, and continue to be, a major influence on the addition of new features to Bliss-11.

The facilities of the Bliss-11 language and compiler had a significant influence on the coding of Hydra.

Some of us believe that Hydra could not have been written in this environment without a language of Bliss's caliber.

Bliss-11 preceded Hydra by too short a time. The unreliability of the compiler during its first year of use hindered kernel development.

Compatibility between Bliss-11 and Hydra was a problem. Changes in Bliss-11 sometimes had unfortunate consequences on Hydra code.

We think these comments reflect the close interdependence between a large programming project (Hydra) and the software engineering tools it uses (Bliss-11). Bliss was in a real sense critical to Hydra's development. The need to debug both Bliss and Hydra simultaneously was a necessary burden.

A common measure, albeit a crude one, of a methodology is the productivity of the programmers which used the methodology. By that measure our development strategy worked very well; the average productivity has been about 20 instructions per man-day for kernel code (the typical industrial average for similar code is 5-7 instructions per man-day).

THE FAILURES

Hardware reliability

Hardware (un)reliability was our largest day-to-day disappointment at the time the evaluation meeting took place. The aggregate mean-time-between-failure (MTBF) of C.mmp/Hydra fluctuated between two to six hours, where a failure is defined to be any situation which triggers the recovery actions described earlier. About two-thirds of the failures were directly attributable to hardware problems.

There is insufficient fault detection built into the hardware.

We found the PDP-11 UNIBUS to be especially noisy and error-prone.

Our paging drums were chosen for their predicted performance, but their reliability was so poor that performance was often a moot point.

The crosspoint switch design is too trusting of other components; it can be hung by malfunctioning memories or processors. [*This almost never happens, but when it does automatic recovery is impossible.*]

We made a serious error in not writing good diagnostics for the hardware. The software developers should have written such programs for the hardware.

In our experience, diagnostics written by the hardware group often did not test components under the type of load generated by Hydra, resulting in much finger-pointing between groups. Faulty hardware is often kept in the user system because only Hydra can provoke and pinpoint errors.

Several components of the system have gone through several development cycles, mostly to improve the handling of exceptional conditions, but we are basically limited by the capabilities of the PDP-11 and its UNIBUS. There appear to be two flaws in many of the off-the-shelf components. One of these was mentioned during the meeting: the lack of mutual suspicion. There are a number of ways in which the entire system can be made to fail if one inessential component does not operate according to specifications. The other flaw was not mentioned: the failure to *contain* errors. Once an error has been detected the goal should be to make absolutely sure that the damage won't spread. Many of the standard components, unfortunately, will "complete" an operation even when an error is known to exist; in completing the operation they destroy data, thus making the error unrecoverable.

There is some good news to report, however. Following the meeting, increased emphasis was given to hardware maintenance. As this paper is written (January 1978) our MTBF has increased to about ten hours and many of the hardware problems seem to be settling out.

The small address space problem

The PDP-11 is a 16-bit minicomputer; of particular interest is the fact that this restricts all addresses generated by a user program to be 16 bits long. These 16 bits can be used to address no more than 64K bytes of memory. We refer to this limitation as the "small address problem", or SAP.

Although we were initially aware that the operating system would have to provide some sort of facility for allowing a user to address more than this amount of memory, we did not appreciate how restrictive the 16-bit limitation would be or to what extent circumventing it would affect performance. Our initial impression was that the 16-bit limitation would be offset by the ability to create multiprocess programs—that the typical program organization would be a larger number of processes, each addressing a smaller amount of memory. That impression turned out to be false, as is reflected in some of the comments made at the meeting:

Our initial prediction that programs would be implemented as small subsystems using less than 64K was wrong.

Multiprocess algorithms do not always produce small programs.

Even though programmers are writing programs which execute on PDP-11's, their tasks are CDC 6600-size.

There is nothing good to say about this problem other than that we were pretty much forced into it.

To circumvent this problem, Hydra provides a facility, supported by the hardware, to divide the address space into 8 pieces, each of which is called a "page." The user is permitted to have an indefinitely large number of pages, but to address only 8 of them at any instant. Operating system facilities are provided to allow the user to dynamically designate which of his pages are to be addressable; he does this by associating a page with one of the 8 "relocation registers" maintained by the hardware. Thus, except that the cost of loading is larger, the addressing scheme is very similar to the use of "base registers" on 360-370 style machines. We have found this facility, however, to be less than ideal.

Page boundaries are absolute, and the programmer must always be aware of them.

The problem is in addressing data. There are easy solutions to addressing code segments.

More relocation registers and a smaller page size would reduce but not eliminate the problem.

We believe the problem would exist even if making pages addressable required no overhead.

Because of the performance penalties associated with managing the address space, the inconvenience cannot be hidden from the user through a high-level language:

L*'s ability to allow access to large amounts of memory has been hindered by the short PDP-11 address. [*L* is a list processing language used for the implementation of large systems.*]

It must be emphasized that not *all* programs are affected by the small address space problem:

In practice, most subsystems have no problem fitting into 64K.

Our failure on the small address problem was really one of misappreciating the way in which the machine would

actually be used. The remark above to the effect that many tasks are 6600-size is a telling one. The machine is comparable in size to a 6600 and people want to use it that way. Big problems often imply big data and we failed to appreciate that during the initial design.

The partitionable system

When we first considered the possibility of building a multiprocessor in 1971, the ability to partition it into several disjoint subsystems was on our list of advantages for such architectures. While we are able to partition processors and memory, we are not able to run Hydra in more than one partition.

C.mmp can be partitioned in such a way that some processors and memories can undergo maintenance and run stand-alone diagnostics without interfering with the larger partition running the operating system.

The primary obstacle to running the operating system in two partitions is the money required to provide each partition with an adequate complement of I/O devices and memory.

We do not know how to provide meaningful communication between the capability structures of the two operating systems.

The principal effect of the failure to meet this goal has been that we must allocate disjoint time for users, hardware maintenance, and operating system testing. At present 28 hours each week are reserved for maintenance. This partitioning has been very inconvenient for all concerned, and has certainly impeded progress on several occasions. Yet it seems clear that we have been unwilling to spend the money necessary to solve the problem—thus it seems safe to conclude that the inconvenience has not been debilitating.

(The lack of) human engineering

As we have mentioned in several contexts previously, the human interface to the C.mmp/Hydra system is not well designed. To some extent this resulted from the novelty of the underlying system structure (we couldn't anticipate some of the kinds of facilities that would be needed by users of either a capability-based or a multiprocessor system). To a large extent, however, the failure seems to have been one of having concentrated on the new, innovative aspects of the system and ignoring more mundane issues.

There is a lack of human engineering in the operating system software which interacts directly with a user sitting at a terminal.

It is difficult to pick up the minimal knowledge needed to know how to do useful things at a terminal.

New users tend to have bad first impressions of the system.

We did not realize how much work was required to make a smooth user interface and so did not allocate enough resources for it.

We suspect the user environment would have received more work had the kernel implementors had to use it during their software development. (All kernel development and maintenance has been done on the PDP-10 computer, which has the Bliss-11 compiler and a linker for C.mmp.)

One particular aspect of the human interface is especially interesting—the command language. It seems to be an almost universal phenomenon that people don't like whatever command language they have used in the past. We were no exception. Thus, rather than modeling our command language on any existing one, we chose to strike out in another direction. In particular, we chose to make the command language a (modest) interactive programming language—with declarations of variables, assignments, conditional and looping control constructs, macros, and so on. The power of this approach seems unquestionable, as is reflected by the following remarks. The remarkable thing (to the editors) is the lack of negative remarks during the meeting; the command language usually comes under heavy attack on other occasions.

The Command Language is much more flexible and powerful than the command scanners found on most systems.

The concept of the Command Language as a programming language was good.

The Command Language user on C.mmp is unique in having complete access to the Hydra environment. Subsystems can almost be implemented directly in the Command Language.

Error reporting by the Command Language is poor.

Another aspect of the human interface is the (lack of a) spectrum of programming languages:

C.mmp lacks the wide range of languages available on conventional systems.

The L* system provides its users with a complete environment compatible with that provided on the PDP-10 by its version of L*.

The L* environment does not seem conducive to the construction of subsystems.

The Algol 68 implementation on C.mmp gives users access to the multiprocess capabilities of C.mmp, but does not yet provide access to capabilities or the Hydra protection environment.

The fact that most subsystem development takes place partially on C.mmp and partially on the PDP-10's (which

have Bliss-11 compilers) is not a severe hindrance now that smooth communication facilities exist between the machines.

It is interesting (to the editors) that the word "baroque" was not used during the meeting; in other contexts it often is. Several features of Hydra and its subsystems do exhibit "second-system-itis". There are things which are more general, and more complicated, than necessary.

Project management

The C.mmp/Hydra project was not a large project by most standards; there were never more than about 15 people, mostly students, working on the project at any one time. Nevertheless we made a number of errors which can only be classified as failures in the management of the project; taken together, these errors constitute one of our largest failures.

Among our errors is a classic! Because the hardware and Hydra structures were new and exciting, we tended to focus on them to the exclusion of the more mundane things which also determine the ultimate utility of any system. This point recurred in many of the points raised at the meeting:

The manpower allocated to the Policy Module was inadequate. In fact this was true of all software outside the kernel.

The failure to stress reliability and performance in the first PM was a mistake.

The user environment was ignored at first because of our natural preoccupation with the Hydra kernel and the research problems it embodied.

We underestimated how much work would be involved in constructing the user environment.

We have a much better idea now about the proper structure (or at least an adequate one) of the user environment than we did when we began building the first subsystems. Implementing basic concepts such as "jobs" and "terminals" in nonpriviledged software has subtle design and reliability implications which we are just now appreciating.

The management style used throughout the project was informal. There were very few memos, formal design reviews, or the other mechanisms of tight management control. In most ways this felt appropriate to the academic environment and the high caliber of the individuals involved. It led to a number of problems, however, and the consensus of the meeting was that the management had been too loose. This is especially evident in the comments relating to a lack of formal specifications and the lack of uniform documentation and coding standards.

The fact that the Hydra implementors did not have to use C.mmp for software development contributed to the neglect of the user environment.

The lack of detailed hardware specifications hindered the parallel development of hardware and software but not the end result.

Software was occasionally developed which took advantage of unspecified "features" of the hardware, making them difficult to change.

Loose management coupled with the modularization technique worked well except in forcing standardization of coding styles.

We should not have depended on graduate students for complete software development for so long. Graduate students cannot keep deadlines reliably and are not tied to the project. [*Furthermore, we feel that Ph.D. students should not spend an inordinate amount of time doing the standard programming chores which characterize any attempt to bring up a complete operating system.*]

Another class of management errors relates to what might be termed "public relations." Being academics, we instinctively react somewhat negatively to the "attention-getting" aspect of PR, forgetting that its "information-providing" function is absolutely necessary. In a number of ways we failed to make information available publicly.

Our problem is basically public relations—performance measurements indicate we have a winner on our hands.

The lack of a smooth user environment was a deterrent to new users which could form the foundation of a happy and vocal user community.

Since Hydra was not easily accessible to people outside the department, we could not adopt a "try it and see" attitude.

Documentation is needed to encourage use internally and generate credibility externally.

A DATA SAMPLER

The previous section concludes our report of the meeting. Since the body of the report contains many subjective and unsubstantiated comments, we decided to include a few examples of the kinds of data on which these comments are based. We have chosen two examples: (1) a study of the effect of the small address problem on a specific user program, and (2) a study of the contention for locks in the Hydra kernel.

A study of the small address problem

The program used in this study of the SAP is HARPY. HARPY is a speech-understanding system which has been implemented on all of the departments major computers: C.mmp, a stand-alone PDP-11 running under UNIX, and the PDP-10 (both KA10, circa 1967, and KL10, circa 1976, processors are available in the department). Since HARPY ex-

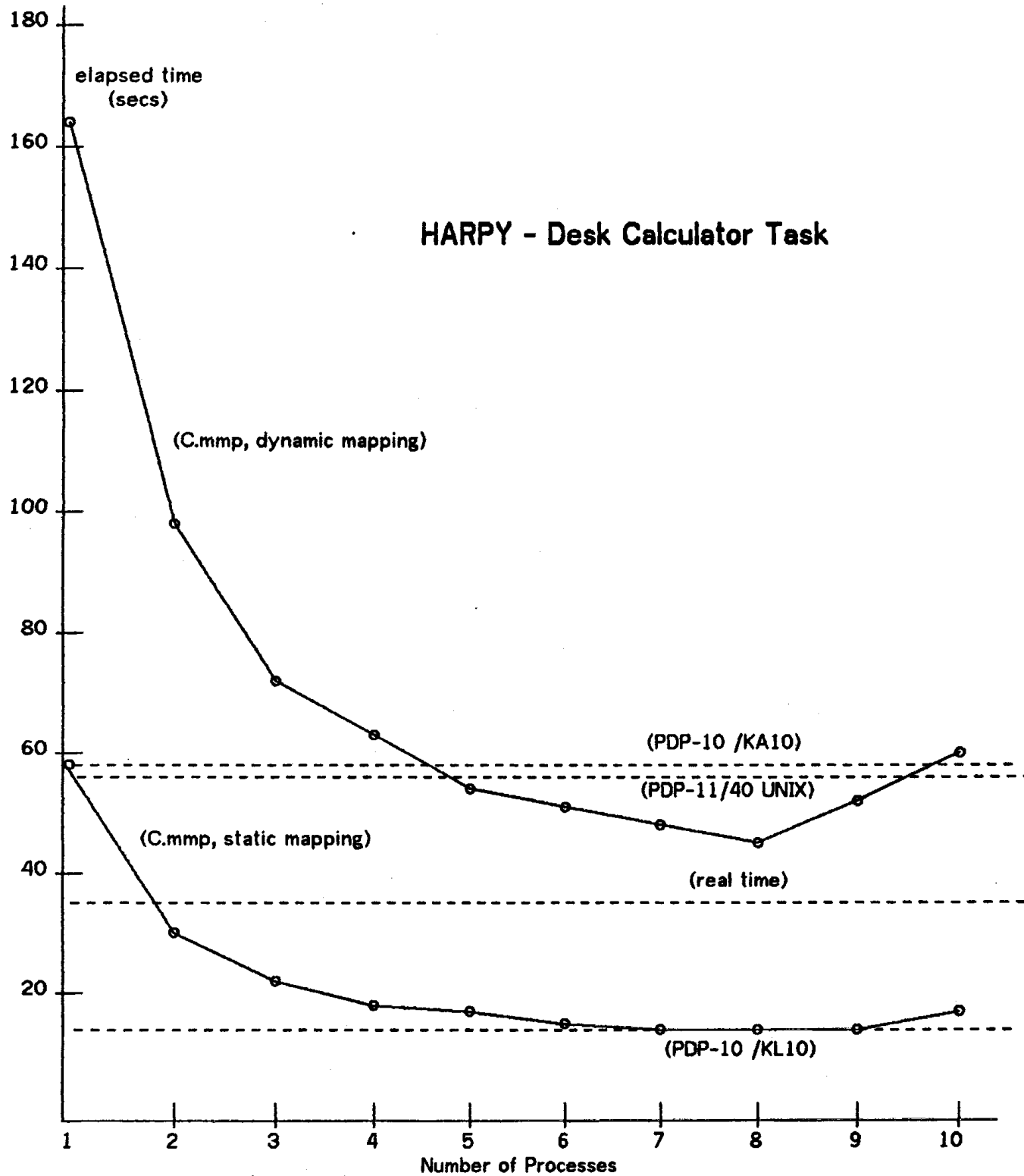

Figure 1—A look at the small address problem

ists on all these machines, it makes a convenient benchmark. (We should point out that HARPY is not necessarily the best application for C.mmp, nor are the HARPY implementations on C.mmp known to be optimal.)

Figure 1 summarizes the data obtained from a series of experiments with HARPY working on a rather small task, namely a voice-input desk calculator that has a 37 word vocabulary.

The horizontal dashed lines represent the performance of single-process implementations of HARPY on the departments uniprocessors. The solid curves represent the performance of two implementations on C.mmp, both of which can utilize any number of processes.

The two HARPY versions on C.mmp differ in their assumptions about the addressability of data. The "static mapping" version knows that all of its data is always addressable, while the "dynamic mapping" version expects to have to do some mapping of relocation registers in order to address the data. In this second version, it must be realized that, in fact, all the data is addressable, and thus no operating system overhead is involved. (The overhead is HARPY checking to see if relocation is necessary—it never is.)

This type of data dramatically illustrates the effect of the SAP on performance—it costs nearly a factor of three in this example. The effect on programming difficulty is at least as great, but is not so easy to illustrate.

Note that the one-process, static mapping version of HARPY runs very nearly as fast as the version running under UNIX, even though the C.mmp version has all the necessary mechanisms for multiprocessing. We think this indicates that the synchronization primitives (spinlocks in shared memory) do not contribute much overhead in this application.

Also note that little improvement in performance is seen beyond three or four processes. This is simply due to a lack of work to do—the small vocabulary simply isn't complicated enough to keep the processors busy. On larger vocabularies we typically see noticeable improvement out to eight processes. The upturn in the curves towards the end is due to the fact that all the faster PDP-11/40 processors are in use. As soon as one PDP-11/20 is used, the whole assemblage of processes slows down. This is because the particular decomposition of the algorithm limits the speed to that of the slowest process.

A study of kernel lock contention

One of the largest potential bottlenecks in a distributed operating system is contention for locks on shared data structures. The hardware monitor has been used to study this; the types of results obtained are shown in Figure 2.

In this study, three programs with seemingly different demands on the system were run while the hardware monitor measured the activity on one processor. The data is illustrative only, since no claim is made that the programs in any way represented a "typical" system load.

The principle result is that it seems we spend consistently less than 1 percent of the time blocked on locks. We do not yet have any measurement of the time lost due to blocking on semaphores.

Static	Program 1	2	3
Total time of measurement (millis)	17393	32924	20255
Number of different locks detected	53	79	181
Average time inside a critical section (micros)	279	378	279
Total number of lock operations	2955	504	4360
Percent of locks which blocked	5.5	11.7	6.1
Percent of time spent in kernel code	61.8	16.9	37.7
Percent of time spent in blocked state	.29	.83	.74

Figure 2—A study of kernel lock contention

CONCLUSIONS

The C.mmp/Hydra project has reached the point at which many of its most interesting and important results will emerge. With a growing user community, increasing reliability and a smoother user interface, we are in a position to gether data on various aspects of system performance under real loads. This data will augment that already collected on isolated algorithms to provide a comprehensive picture of C.mmp/Hydra performance. Along the way to constructing the current system we managed, in our opinion, to do some things well and some things not so well. This paper has been our attempt to report those opinions in the hope that others may benefit from our experiences.

ACKNOWLEDGMENTS

A large fraction of the faculty and staff of the Computer Science Department at CMU have been involved with C.mmp and Hydra over the past five years—as designer/implementors, as users, or as constructive critics. We are deeply indebted to all of them. We are especially indebted, however, to those who participated in the meeting that is reported here:

Hardware:	Bill Broadley, Jim Teter
Hydra:	Sam Harbison, Dave Jefferson, Roy Levin, Hank Mashburn, Fred Pollack
Non-kernel OS:	Bill Corwin, Rick Gumpertz
Performance Evaluation:	Sam Fuller
Major Users:	Anita Jones, Bruce Leverett, Pete Oleinick, George Robertson
Others:	Joe Newcomer, Bill Wulf

We would also like to thank Guy Almes, Peter Schwarz, and the NCC '78 referees for their helpful suggestions for this paper.

C.mmp/Hydra BIBLIOGRAPHY

This bibliography includes references to papers, articles, and theses related to the design, development, and measurement of C.mmp and Hydra. There are numerous internal documents and memos which are not included.

1. Almes, G. and G. Robertson, "An Extensible File System for Hydra," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa. 1978. (This paper will appear in the *Proceedings of the Third International Conference on Software Engineering*, 1978.)
2. Bell, C. G., W. Broadley, W. A. Wulf, and A. Newell, "C.mmp: The CMU Multiminiprocessor Computer: Requirements and Overview of the Initial Design," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., August 1971.
3. Bhandarkar, D. P., "Analytic Models for Memory Interference in Multiprocessor Computer Systems," Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., September 1973.
4. Bhandarkar, D. P. and S. Fuller, "Markov Chain Models for Analyzing Memory Interference in Multiprocessors," *ACM/IEEE First Annual Symposium on Computer Architecture*, Dec. 1973, pp. 231-239.
5. Cohen, E., "Problems, Mechanisms and Solutions," Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August 1976.
6. Cohen, E. and D. Jefferson, "Protection in the Hydra Operating System," *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975, pp. 141-160.
7. Fuller, S. and P. Oleinick, "Initial Measurements of Parallel Programs on a Multi-miniprocessor," *IEEE CompCon'76*, September 1976, pp. 358-363.
8. Fuller, S., "A Cost/Performance Comparison of C.mmp and the PDP-10," ACM/IEEE Symposium on Computer Architecture, Jan. 1976.
9. Fuller, S., Swan, R. and W. A. Wulf, "The Instrumentation of C.mmp: A multi-(mini)-processor," *IEEE CompCon'73*, 1973, pp. 177-180.
10. Fuller, S. H., and D. K. Stevenson, "The Performance Monitor for C.mmp," *11th Annual Allerton Conference*, Urbana, Illinois, October 1973.
11. Fuller, S. H., "Recent Developments in Multiprocessor Computer Systems," *CALCOLO*, Vol. XII, No. 1, June 1975, pp. 35-58.
12. Jones, A. K. and W. A. Wulf, "Toward the Design of Secure Systems," *Software—Practice and Experience*, Vol. 5, 1975, pp. 321-333.
13. Levin, R., E. Cohen, W. Corwin, F. Pollack, and W. A. Wulf, "Policy/Mechanism Separation in Hydra," *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975, pp. 132-140.
14. Marathe, M., and S. H. Fuller, "A Study of Multiprocessor Contention for Shared Data in C.mmp," *ACM SIGMETRICS Conference*, Washington, D.C., December 1977.
15. Newcomer, J., E. Cohen, W. Corwin, D. Jefferson, T. Lane, R. Levin, F. Pollack, and W. Wulf, "Hydra: Basic Kernel Reference Manual," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., 1976.
16. Newell, A., P. Freeman, D. McCracken, and G. Robertson, "The Kernel Approach to Building Software Systems," Computer Science Research Review 1970-71, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., 1971.
17. Newell, A., D. McCracken, and G. Robertson, "L*: An Interactive, Symbolic Implementation System," Department of Computer Science Technical Report, Carnegie-Mellon University, October 1977.
18. Newell, A. and G. Robertson, "Some Issues in Programming Multi-Mini-Processors," in *Behavior Research Methods and Instrumentation*, Vol. 7, No. 2, March 1975, pp. 75-86.
19. Oleinick, P. H., and S. H. Fuller, "The Implementation and Evaluation of a Parallel Algorithm on C.mmp," Department of Computer Science Technical Report, Carnegie-Mellon University, December 1978.
20. Reid, B. K. and J. Newcomer, ed., "The Hydra Songbook—A Vigilante User's Manual," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., October 1975.
21. Reiner, A., and J. Newcomer, ed., "Hydra User's Manual," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., August 1977.
22. Strecker, W. D., "An Analysis of the Instruction Execution Rate in Certain Computing Structures," Ph.D. Dissertation, Carnegie-Mellon University, 1971.
23. Wulf, W. A. and C. G. Bell, "C.mmp—A Multi-mini-processor," *Proceedings of the Fall Joint Computer Conference*, 1972, pp. 765-777.
24. Wulf, W. A. and R. Levin, "A Local Network," *Datamation*, February 1975.
25. Wulf, W. A., "Reliable Hardware-Software Architecture," *Proceedings of the International Conference on Reliable Software*, Los Angeles, 1975.
26. Wulf, W. A., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The Kernel of a Multiprocessor Operating System," *CACM* 17, 6, June 1974, pp. 337-345.
27. Wulf, W. A., R. Levin, and C. Pierson, "Overview of the Hydra Operating System," *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975, pp. 122-131.