C.mmp

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-

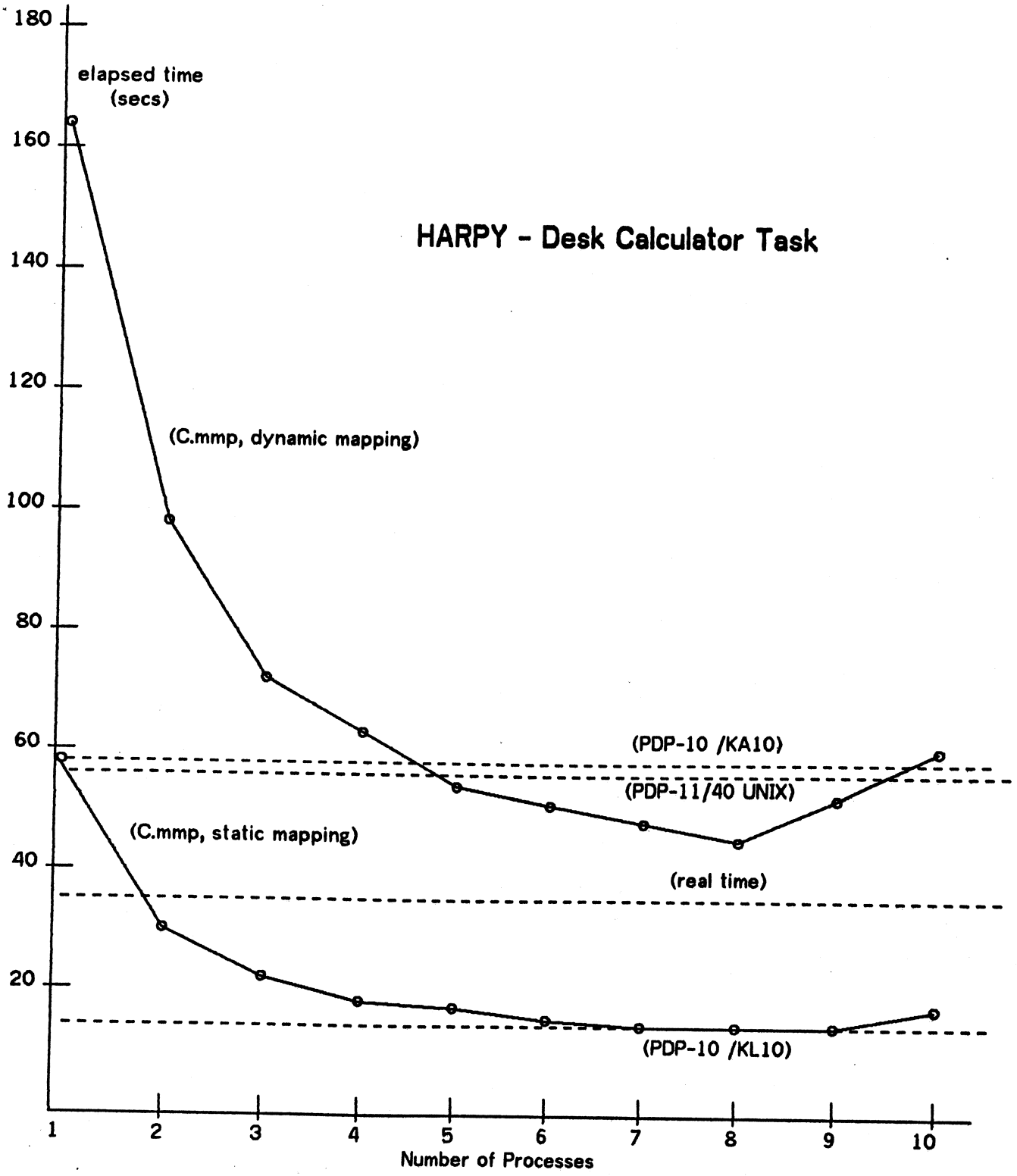


Figure 1—A look at the small address problem

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.

C.mmp—A multi-mini-processor*

by WILLIAM A. WULF and C. G. BELL

Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION AND MOTIVATION

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

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

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

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

REQUIREMENTS

The CMU multiprocessor project is designed to satisfy two requirements:

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

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

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

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

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

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

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

TIMELINESS OF MULTIPROCESSOR

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

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

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

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

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

RESEARCH AREAS

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

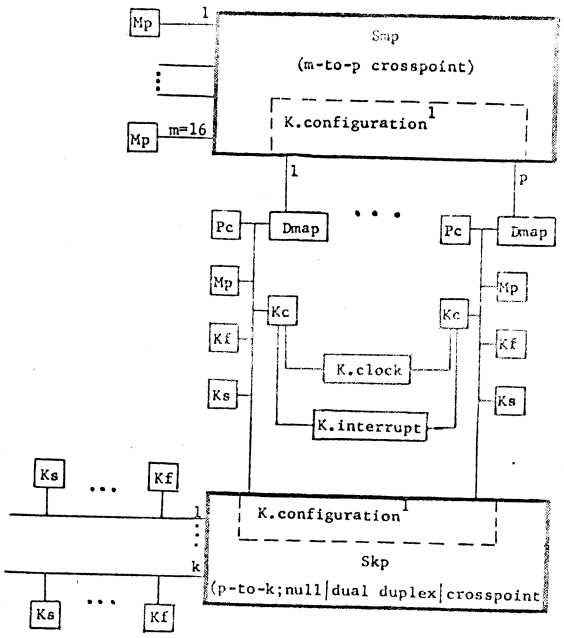
1. The multiprocessor hardware design which we call the PMS structure (see Bell and Newell, 1971). Few multiprocessors have been built, thus each one represents an important point in design space.
2. The processor-memory interconnection (i.e., the switch design) especially with respect to reliability.

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

THE HARDWARE STRUCTURE

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

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



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

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

Figure 1—Proposed CMU multiminiprocessor computer/C.mmp

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

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

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

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

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

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

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

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

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

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

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

Dmap

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

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

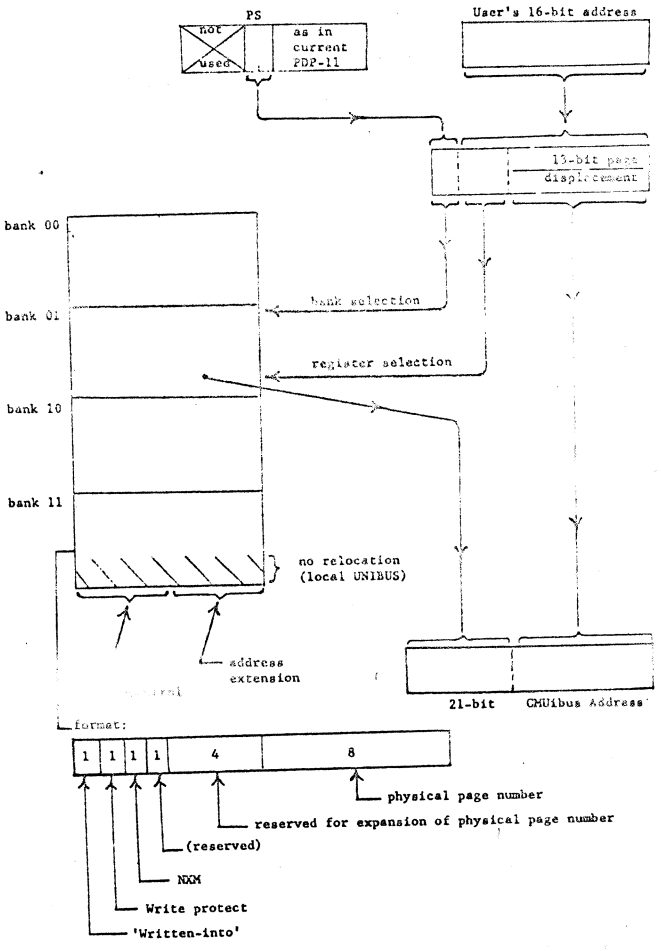


Figure 2—Format of data in the relocation registers

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

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

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

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

THE MEMORY INTERFERENCE PROBLEM

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

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

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

Strecker gives the following relations:

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

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

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

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

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

Given data of the form in Figures 3 and 4 it is possible to obtain the cost effectiveness of various processor-memory configurations. An example of this information for a particular memory configuration (16 memories, $t_c = 400$) and three different processors (roughly corresponding to three models of the PDP-11 family) is plotted in Figure 5. Note that a small configuration of five Pc.1's [illegible] performance of 4.5×10^6 accesses/second ([illegible] of such a system is approximately $37[illegible], [illegible] a cost-effectiveness of 12. Replacing these five processors with the same number of Pc.3's yields a UER of 15×10^6 for about $625K, or a cost-effectiveness of about 24. Following this strategy provides a very cost-effective system once a reasonably large number of processors are used.

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

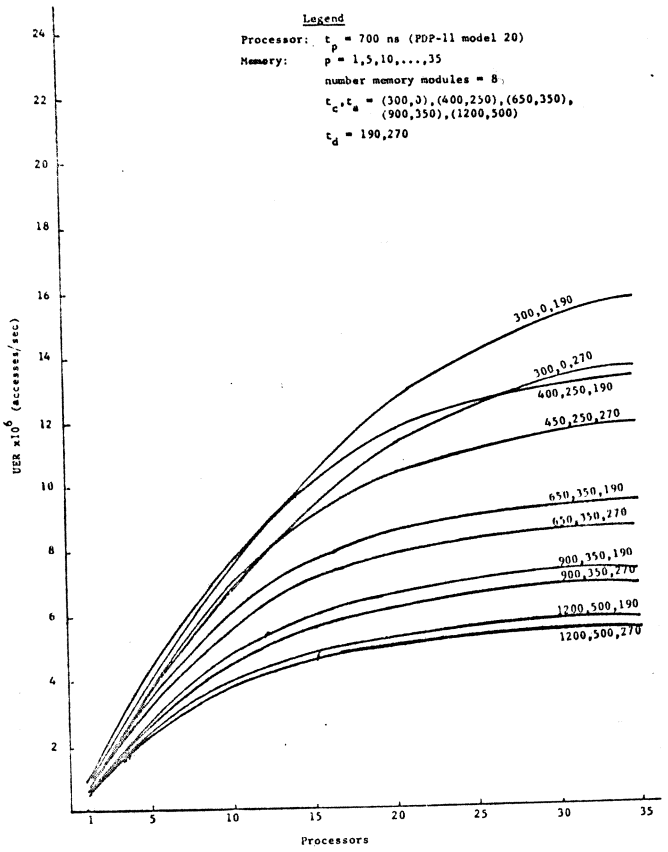


Figure 3—Performance for various memory-processor configurations

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

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

Software page-placements

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

Switch, Smp, measurement

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

A cache

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

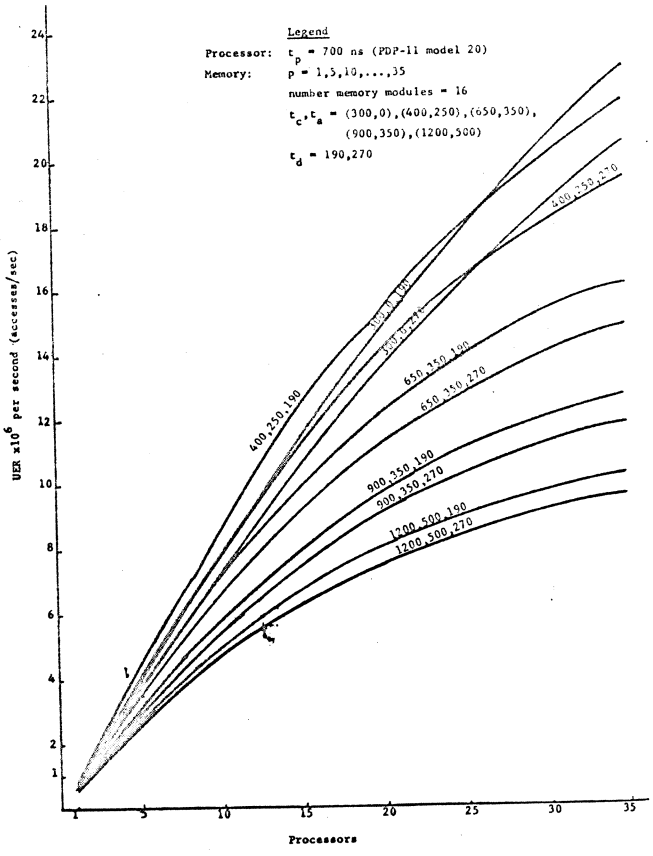


Figure 4—Performance for various memory-processor configurations

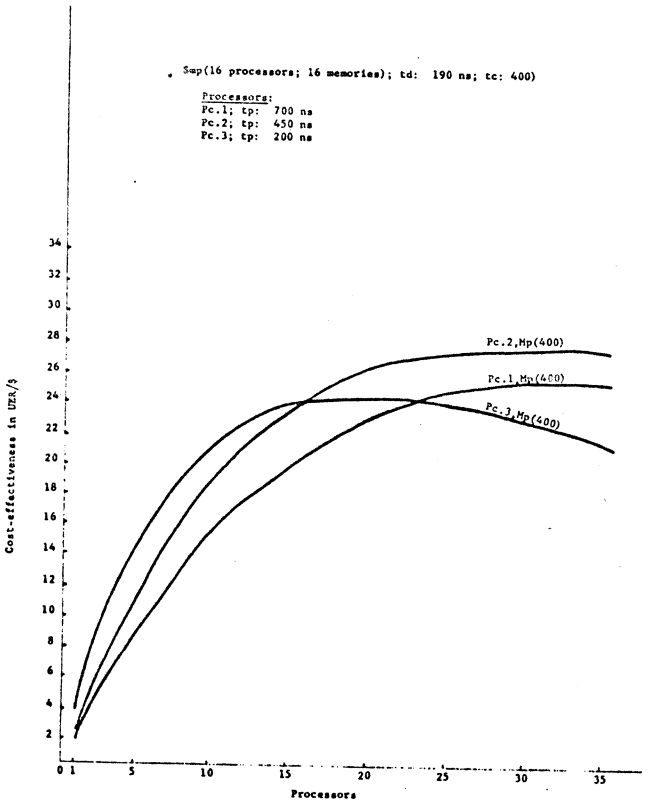


Figure 5—Cost effectiveness (UER/$)

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

By introducing such a cache, however, a potential problem is created regarding the validity of data since it might be possible to have sixteen different values of a single variable at a given instant of time. A scheme for avoiding this is to allow only information from "read only" pages (especially instructions) to appear in the cache. (In particular, the bit marked 'reserved' in Figure 2 is used to signal that data from the page may be placed into the cache.) Traces of PDP-11 programs executions indicate that a small cache (256–512 words) will capture 70–80 percent of the eligible references and 40–50 percent of all references. McCredie (1972) has studied the effect of such a cache on overall system performance both analytically and by simulation. The results of these studies indicate an improvement of 10–40 percent in overall system performance.

THE OPERATING SYSTEM

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

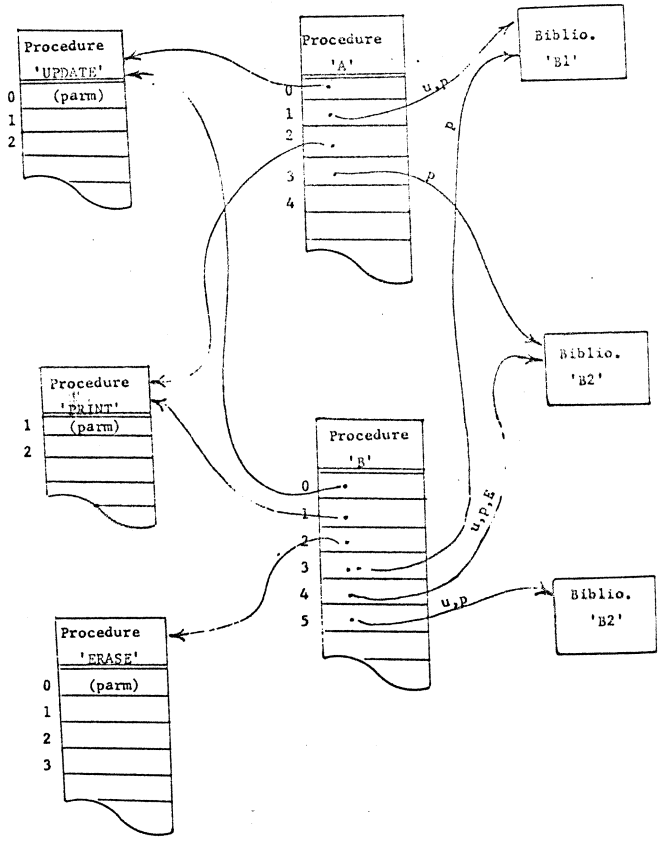


Figure 6—Example of LNS mapping and protection

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

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

THE RELIABILITY PROBLEM

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

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

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

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

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

Redundancy

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

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

Consistency

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

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

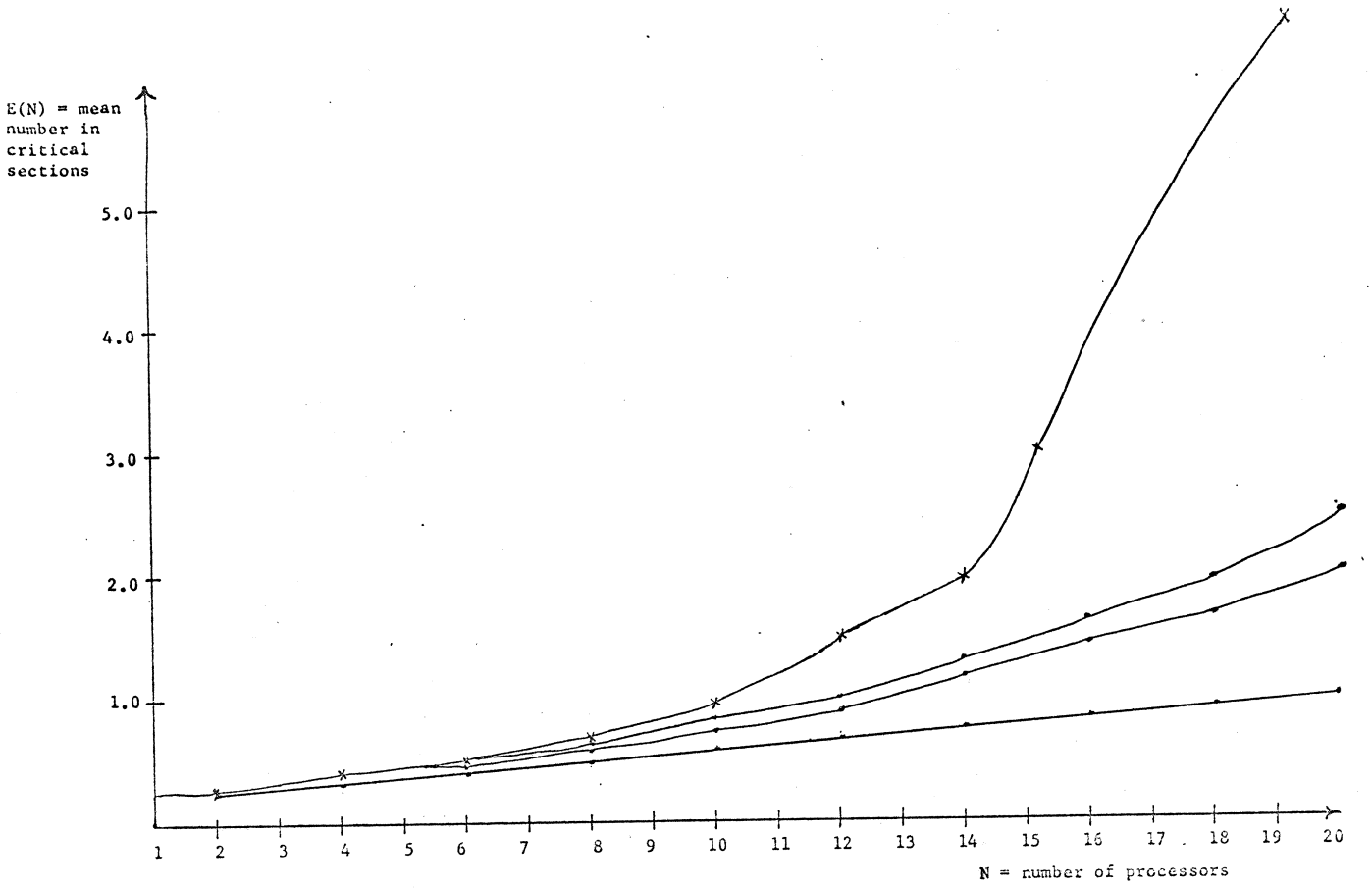


Figure 7—Mean response time for scheduling

Diagnostics

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

THE LOCK PROBLEM

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

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

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

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

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

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

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

PROGRAMMING ISSUES

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

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

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

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

Language issues

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

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

Algol-like scope rules

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

Encoding

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

Fixed representations

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

SYSTEMS ISSUES

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

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

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

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

SUMMARY

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

ACKNOWLEDGMENT

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

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

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

REFERENCES

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

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

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

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

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

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

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

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

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

10 E DIJKSTRA
Structured programming
Software Engineering October 1969 Rome

11 R KRUTAR
Personal Communication 1971

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

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

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

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

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

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

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

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

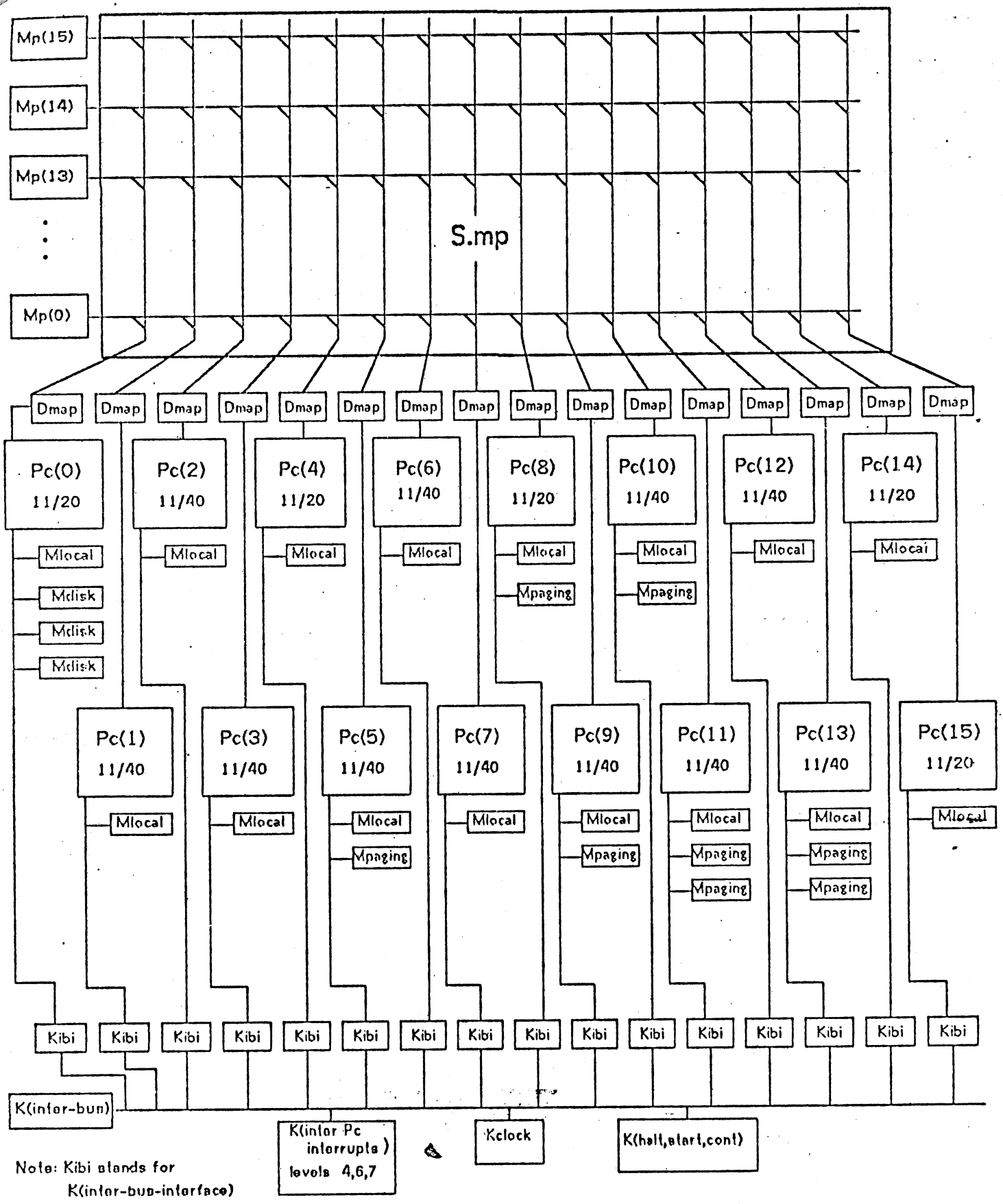


Fig. 1.1 PMS diagram of C.mmp

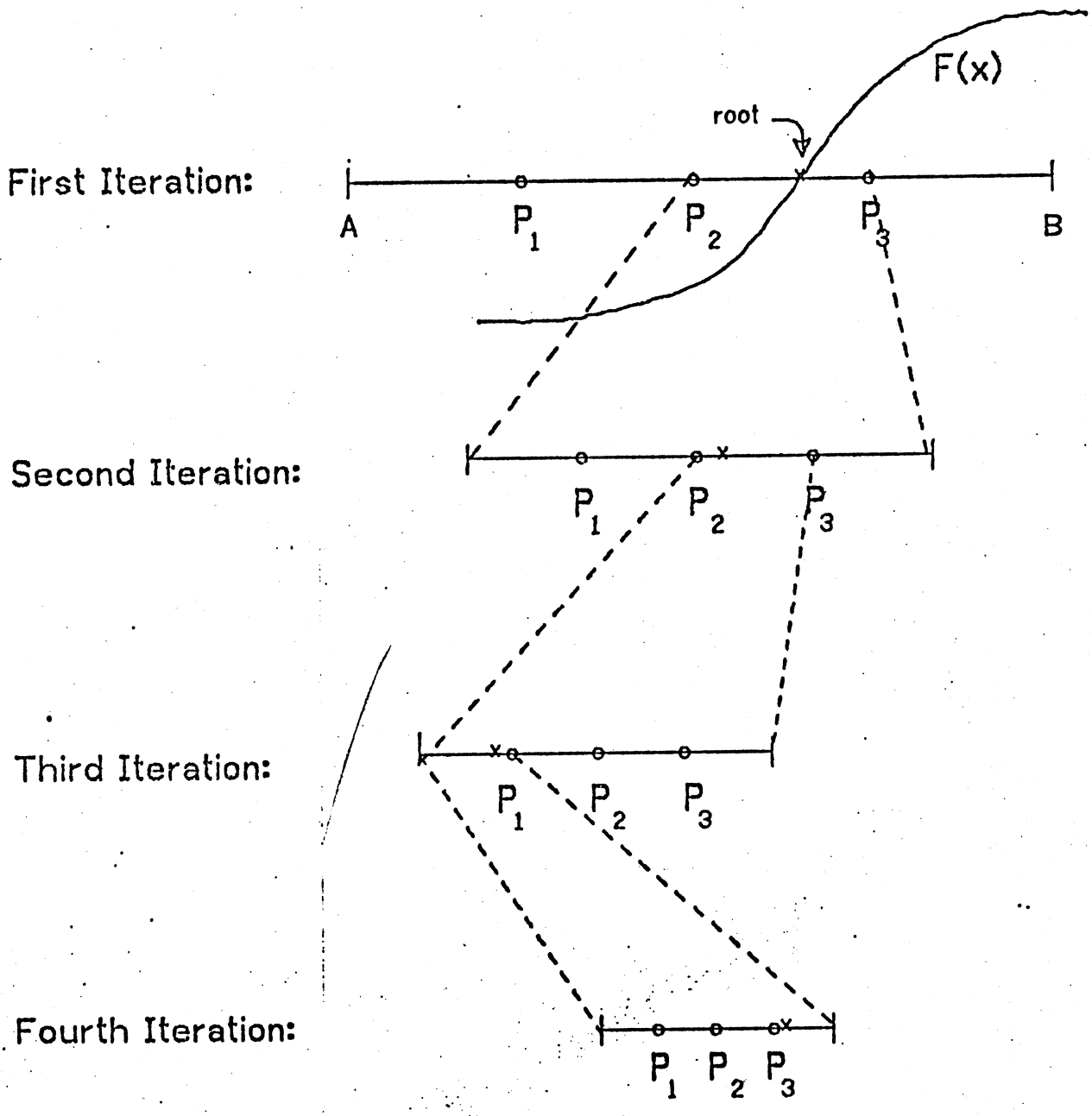


Figure 2.1 Rootfinding Program Using Three Processors

THE IMPLEMENTATION AND EVALUATION OF A PARALLEL ALGORITHM ON C.mmp

(Preliminary)

P.N. Oleinick and S.H. Fuller

January 18, 1978

TABLE OF CONTENTS

Figures and Tables

1. Introduction

Most papers that extol the virtues of multiprocessor computer systems cite the higher throughput [Sauer 1977], and cost/performance [Fuller 1976] over the more traditional uniprocessor. However, both of these performance advantages can be realized only if the software effectively exploits the parallelism in the machine. To date, the task of writing effective parallel software is still an ad-hoc procedure of constructing code for a one of a kind machine. Since multiprocessors are almost as different from one another as they are from uniprocessors it is difficult to apply insight gained from writing parallel software for one multiprocessor to another totally different machine. Yet by documenting the performance of various implementations of several algorithms on one machine we can shed some light on how effective various strategies are at capturing parallelism.

The purpose of this paper then is to provide a first-hand look at the implementation of parallel algorithms on a multiprocessor. The nature of this investigation is experimental rather than theoretical in that the results we present are derived from the measurement of real programs running on a real multiprocessor - C.mmp.

The basic structure of C.mmp, as shown in the PMS diagram of Figure 1.1 is that of the canonical multiprocessor. A detailed description of C.mmp is provided in the original article on C.mmp by Bell and Wulf [1972], but the following description should provide a sufficient background for this investigation.

C.mmp is organized as a system of 16 central processors (Pc's) that share a centrally located large primary memory that presently consists of 2.5 Megabytes. The 16 Pc's are completely asynchronous computing elements: 5 are PDP-11/20's and the remaining 11 are PDP-11/40's. They are connected to the shared primary memory via a 16 x 16 crosspoint switch. The operation of the switch is similar to a 16 ported memory in that up to 16 memory transactions can be performed simultaneously. I/O devices, unlike memory, are associated with an individual processor. Thus for example, an I/O request to a device on Pc[0], perhaps a disk, is performed by the requesting Pc sending an interprocessor interrupt to Pc[0] causing initiation of the appropriate I/O interrupt service routine on Pc[0].

Hydra is C.mmp's general-purpose multiprogramming operating system [Wulf *et al.*, 1974; Wulf *et al.*, 1975; Levin *et al.*, 1975]. It is a collection of basic or *kernel* mechanisms of "universal applicability" and "absolute reliability." Upon this solid core an arbitrary number of systems created from these mechanisms can co-exist simultaneously. *Hydra* is organized as a set of re-enterant procedures that can be executed by any of the processors. In fact, several processors can simultaneously execute the same procedure. This concurrency is accomplished by placing "*locks*" around the operating system's critical data structures. These locks maintain mutual exclusion where necessary. Throughout this paper we will refer to *Hydra* as the *Kernel* or the Operating System.

In the following sections we develop a parallel algorithm to be used as a case study and derive its theoretical performance. We enumerate the contributions to

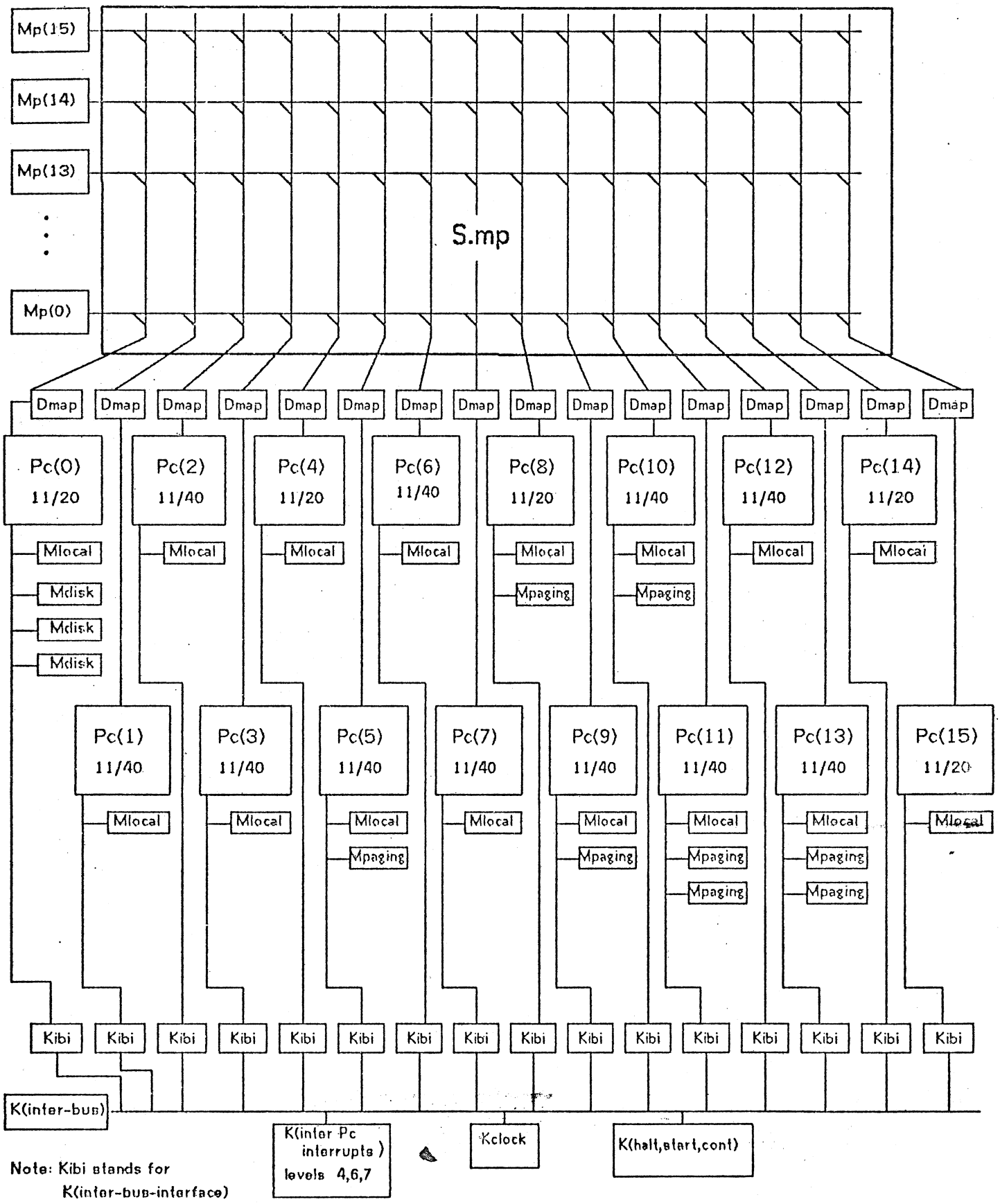


Fig. 1.1 PMS diagram of C.mmp

performance fluctuation and degradation from several sources and quantify the magnitude of each source in terms of the program's performance. One dominant influence on performance is the process synchronization mechanism. We compare several alternative synchronization mechanisms and conclude with a graph showing the range of inter-synchronization times for which each mechanism is preferable.

2. Description of the Rootfinding Algorithm

The purpose of this study is to present quantitative performance results for implementing parallel algorithms on a multiprocessor. Rather than attempting to measure a broad spectrum of problems we have chosen to study various implementations of a single problem in order to observe and measure in depth the performance tradeoffs in the implementation process.

Two criteria that our case study problem had to meet were: the problem must be nontrivial enough to have interesting implementation tradeoffs and low enough complexity to permit the focus of attention on implementation issues rather than algorithm issues. The candidate problem we finally selected is the rootfinding task.

We have chosen to consider this problem not because it particularly well-suited for parallel solution, but rather because it is a relatively straight forward task that requires a significant amount of inter-process communication. According to Stone[1973], algorithms like the rootfinding algorithm that exhibit speed-up gains proportional to the logarithm of the number of processes fall into a class of problems at best considered poor candidates for parallel processing. However, the underlying control structure present in this procedure, that of the synchronous parallel algorithm, is representative of many parallel decompositions of otherwise serial algorithms. For this reason it is worthwhile to understand the nature of the control structure and to study the influences on its performance. Investigations now in progress are considering larger problems and alternative control structures better able to exploit the available parallelism of C.mmp [Oleinick 1978].

Specifically we will consider the problem of finding the root of a monotonically increasing function in a bounded region. If we assume no special information about the behavior of the function, the best procedure for a uniprocessor under these circumstances is a binary search. An obvious decomposition of the binary search into n parallel processes on a multiprocessor is to evaluate the function simultaneously at n equidistant points within the bounded region.

The optimal placement of the n processes on the interval is known [Kung 1976], but to minimize the complexity of the algorithm in order to focus on the synchronous control structure we will use the sub-optimal (but good) technique illustrated in Figure 2.1. The n parallel processes perform function evaluations at the n points that divide the interval into $n+1$ equal subintervals. Since our function, $F(x)$, is a monotonic function, the sub-interval that contains the root is the sub-interval with opposite signs for $F(x)$ at its end points. The other sub-intervals are discarded and the procedure repeats this basic iteration until one of the function evaluations is within ϵ (i.e. an acceptably small interval close to zero) of the zero-crossing.

For the measurements presented here the function we are evaluating is the normal integral:

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty \exp(-1/2t^2)\ dt \quad - \quad h \qquad (2.1)$$

For $x < 2.32$ the following truncated power series was used to evaluate $F(x)$:

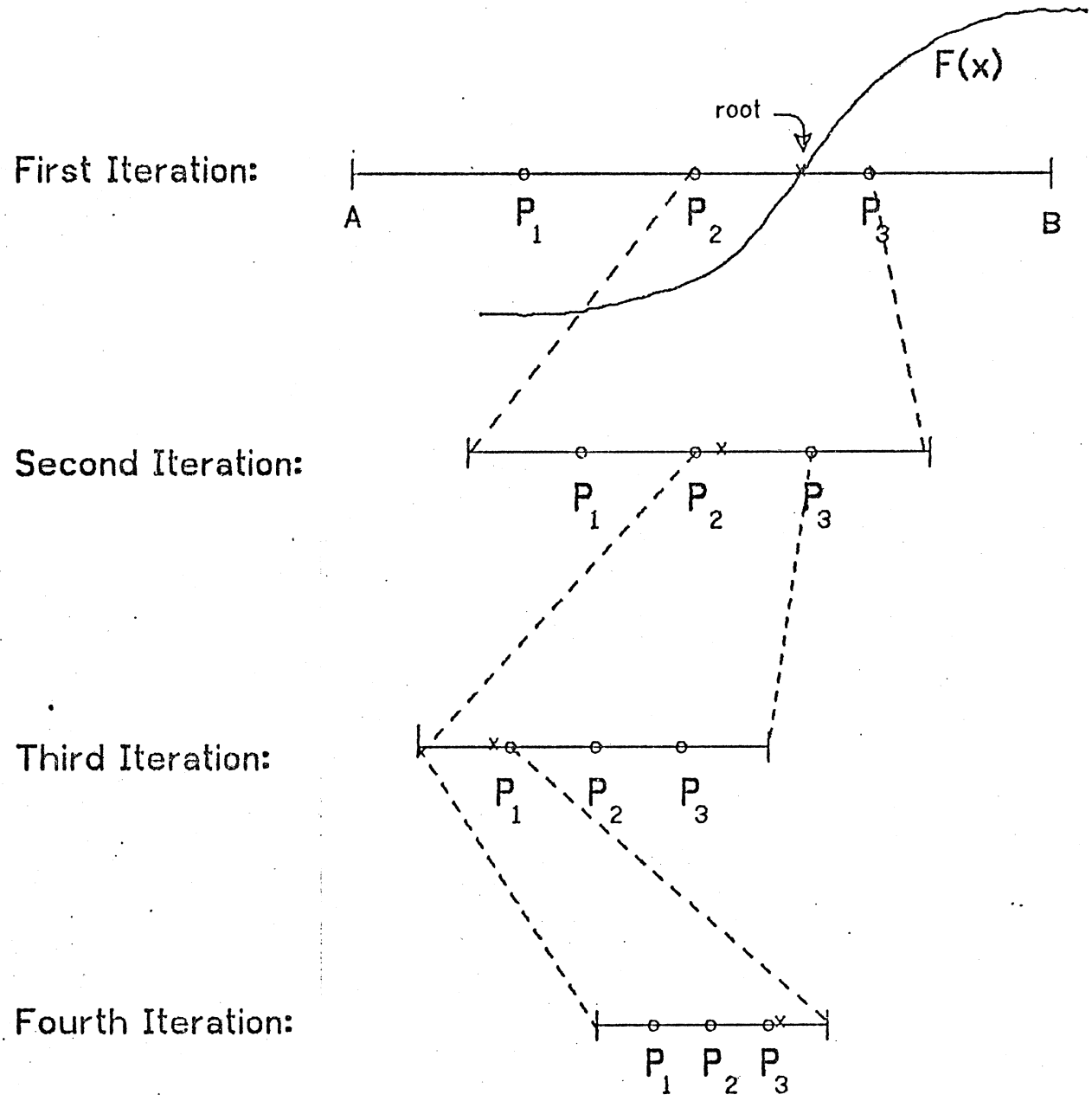


Figure 2.1 Rootfinding Program Using Three Processors

$$\left(x + \frac{x^3}{3} + \frac{x^5}{3*5} + \frac{x^7}{3*5*7} + \frac{x^9}{3*5*7*9} + \ldots \right) - h \qquad (2.2)$$

and for larger x we used the continued fraction:

$$1/(x+1/(x+2/(x+3/(x+\ldots)))) - h \qquad (2.3)$$

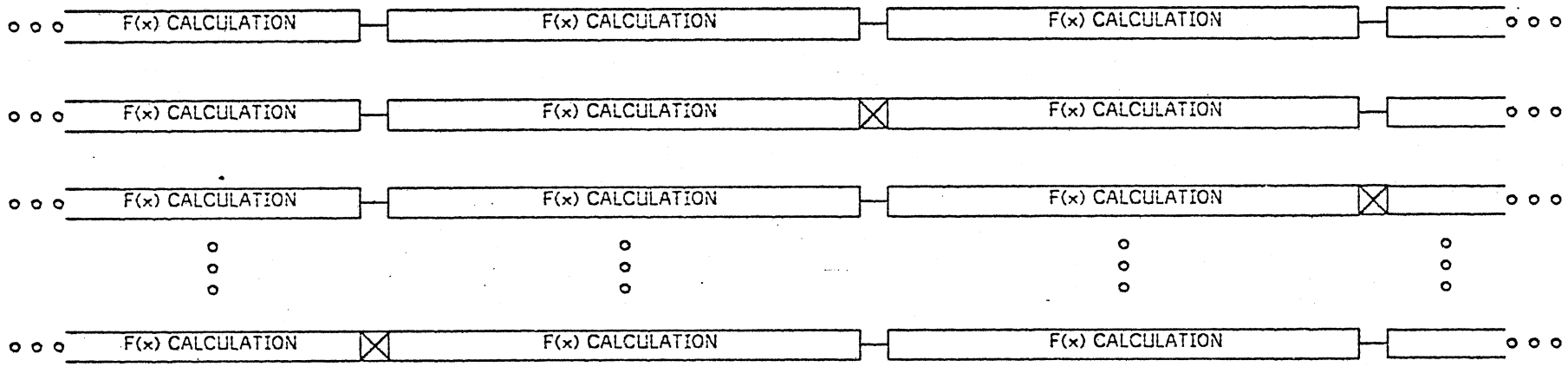
We selected this normal integral because it is an important transcendental function that exhibits two characteristics important to our measurement studies: it requires a nontrivial amount of computation, and the type and length of computation are data dependent.

In order to evaluate the performance of our implementations of the rootfinding algorithm we first calculate the theoretical, or overload-free, performance curves.

The basic cycle in the rootfinder is the independent evaluation of the function by each of the cooperating processes and, upon finishing, the communication of each process with the other processes by posting the results of its function evaluation. Notice that the new interval is not located until all of the processes have posted their results[1]. When the last process finishes its function evaluation it assumes the jobs of finding the new root-containing interval and *waking up* all of the waiting processes. This basic cycle we will call a STAGE.

Under ideal conditions the cooperating processes in the rootfinder would exhibit the execution behavior found in Figure 2.2. Each process performs a function evaluation independently. They all finish at the same instant in time and after a very brief bookkeeping calculation, perform a new $F(x)$ calculation, on an interval reduced by $1/(n+1)$. In practice, we seldom find this to be the case. The fluctuations in performance stem from sources intrinsic to the multiprocessor as well as the rootfinding program. In the next section we investigate these sources of fluctuation.

[1]Obviously the new interval is located as soon as the sub-interval is bounded but again we have opted for a more straight-forward algorithm in order to focus on the implementation issues.



⊠ = LOCATING THE INTERVAL THAT CONTAINS THE ZERO-CROSSING AND REDISPATCHING THE N PROCESSES

Figure 2.2 Optimal Performance of the Rootfinding Algorithm

3. Sources of Performance Fluctuation

3.1 Introduction

In this case study there are three distinct sources of performance fluctuation: the variation in the amount of computation required to perform a function evaluation, the individual hardware elements' performance characteristics, and the operating system. We will identify the nature and measure the magnitude of each of these sources starting with the variation in the *F(x)* calculation as it is the most straight forward of the three.

3.2 The Variation in the F(x) Calculation

The elapsed time to perform a function evaluation is data dependent. The distribution of the compute time is shaped approximately Normal as shown is Figure 3.1. The mean is about 100 milliseconds with almost an equal number of samples on each side of the mean[2]. Thus we might model the expected finishing time for a process performing an *F(x)* calculation to be a random variable with a Normal distribution. As other functions would have other compute time distributions, we derive the theoretical performance for the constant and exponential cases also.

Let the time taken by the *ith* stage in the rootfinding procedure be the random variable T_i. Since all of the processes are performing the same calculation, each process executes for a random amount of time, *t* (see figure 3.2), taken from some distribution. Because all of the processes must finish their function evaluations before the new sub-interval is located

$$T_i = MAX(t_1, t_2, t_3, \ldots , t_n) \tag{3.1}$$

From elementary order statistics the expected value of the largest order statistic in random samples of *n* from a parent distribution with continuous strictly increasing cdf P(x) is

$$E(X_{(n)}) = \int_{-\infty}^{\infty} nx[P(x)]^{n-1} dP(x) \tag{3.2}$$

If we know nothing about the distribution of the t_i other than the mean u and standard deviation s, the expected value of the largest order statistic T_i, reduces to

$$E(T_i) \leq \mu + \frac{n-1}{\sqrt{(2n-1)}} * \sigma \tag{3.3}$$

This bound can be replaced in the exponential case by the equality

[2] On an 11/20 processor

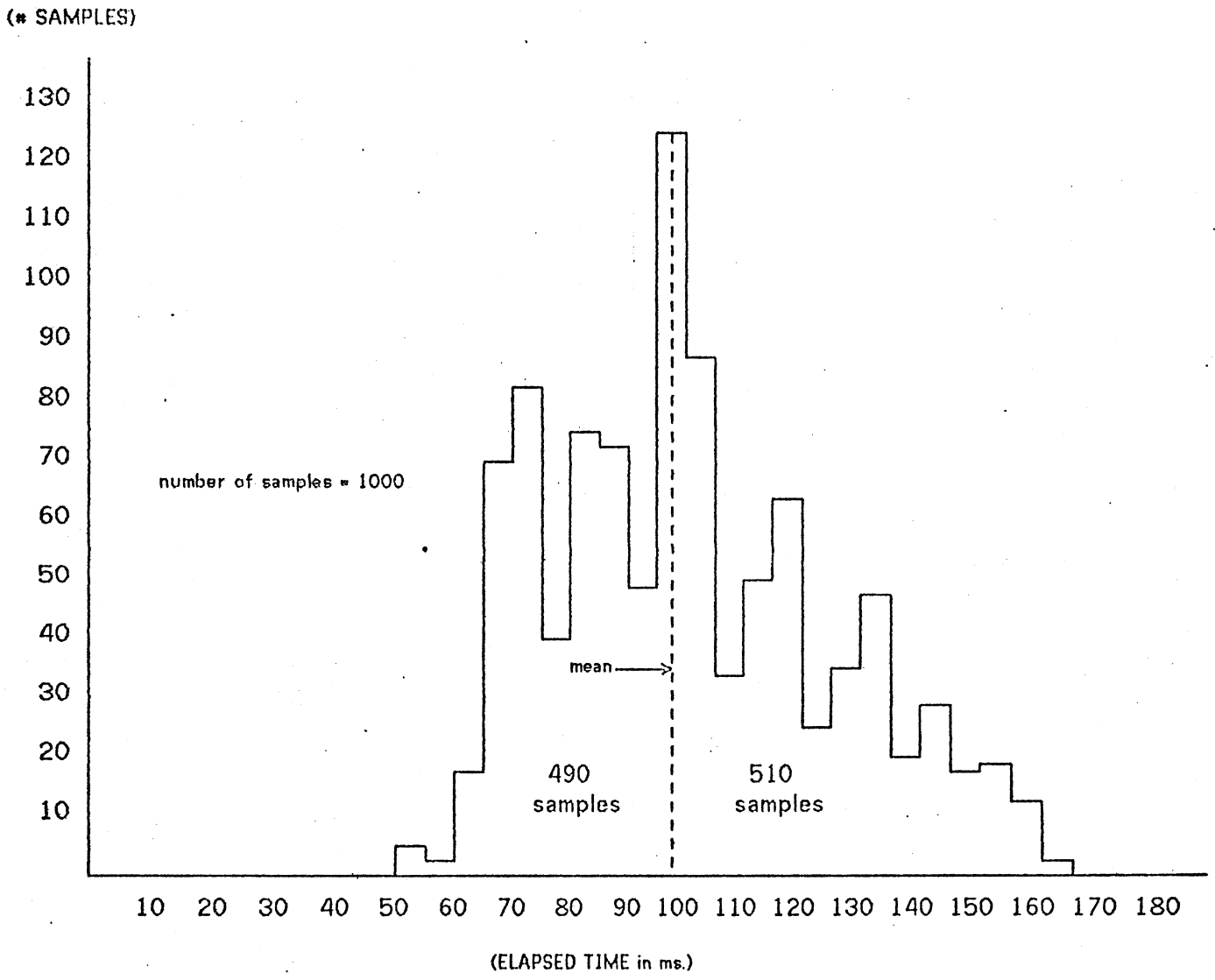
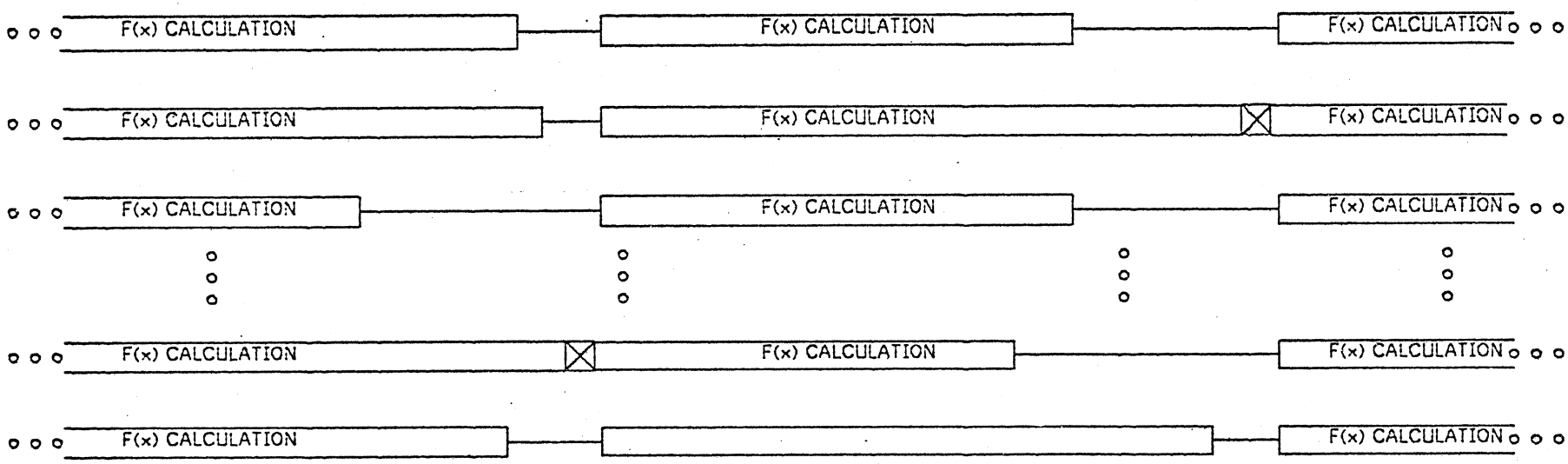


Figure 3.1 Distribution of the Time to Calculate F(x)



⊠ = LOCATING THE INTERVAL THAT CONTAINS THE ZERO-CROSSING AND REDISPATCHING THE N PROCESSES

Figure 3.2 Performance Degradation Due to Variation in the F(x) Compute Time

$$E(T_n) = n\mu \sum_{j=0}^{n-1} \binom{n-1}{j} \frac{(-1)^j}{(j+1)^2} \qquad (3.4)$$

For the Normal case we consult Teichroew's[1956] tables for the expected value of the largest order statistic drawn from the $N(0,1)$ distribution.

When the expected value of the compute time is a constant, equation 3.3 is replaced by the simple equality $E(T_i) = u$.

If we are interested in the performance speedups obtained when we put more processes to work finding roots, we need to estimate the average time to locate a root as a function of the number of processes. Since every iteration in the rootfinding procedure reduces the interval of uncertainty, L, by a factor of $n+1$ it takes $Ceiling(\text{Log}_{n+1} L)$ iterations to locate the root in a bounded interval of length L. Thus in our example let R_i denote the number of iterations necessary to arrive within ϵ of the root using i processes. For our choice of ϵ, R={54, 34, 27, 23, 21, 19, 18, 17, 16, 16, 15, 15,...} iterations. Notice that it takes the same number of iterations to locate the root using nine and ten or eleven and twelve processes. This is because the number of iterations must be an integer. Thus, there is little to be gained by incorporting many processes in the procedure. In this study the maximum number of processes we will use is nine.

We can estimate the runtime of the rootfinder to be the following:

$$\text{Runtime}(n) = \sum_{k=1}^{R_n} T_k = R_n * E(T_n) \qquad (3.5)$$

Often we will be interested in the speedup achieved through parallelism. We will use the following formula to calculate speedup:

$$\text{Speed up}(n) = \frac{\text{Runtime}(1)}{\text{Runtime}(n)} \qquad (3.6)$$

Figure 3.3 is a plot of the speedup *vs.* number of processes for the following three distributions:

Distribution	u	s
Constant	1000	0
Normal	1000	278
Exponential	1000	1000

The glitches in the curves are a result of the *Ceiling* function in the equation for the number of iterations to perform. Because the number of iterations must be an integer value, the curves are not smooth.

This figure contains calculated no-overhead performance curves for three sample $F(x)$ distributions with standard deviations ranging from 0 *to* u. The

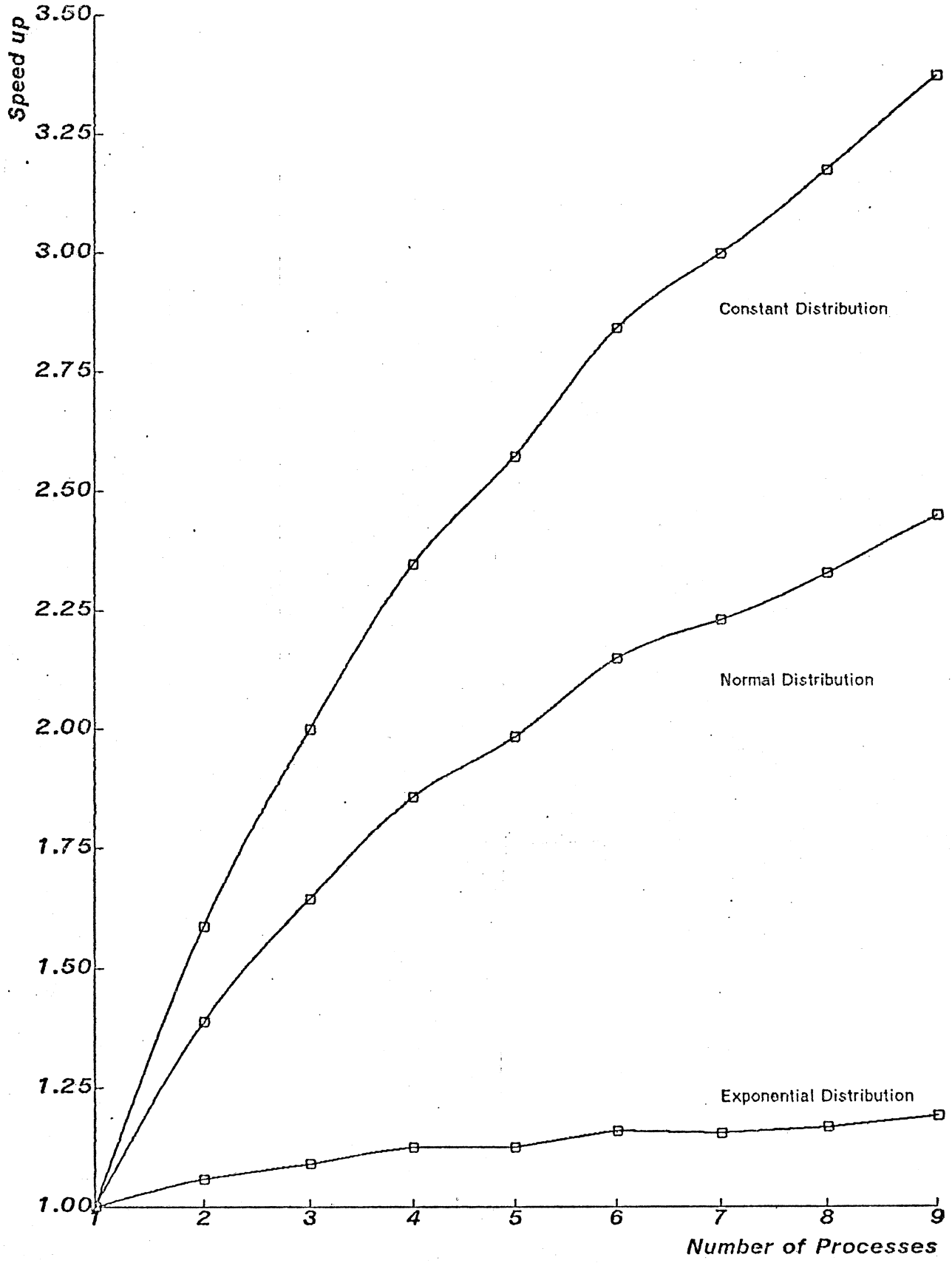


Figure 3.3 Speed up vs. Number of Processes for Ideal Multiprocessor

performance range is from negligible speedup when the compute time for the function evaluation is exponentialy distributed to more than a factor of 3.3 speedup for nine processes when the distribution of the $F(x)$ calculation is a constant. The Normal curve between these extremes closely approximates the actual $F(x)$ distribution and is included for comparison.

Another way to view this data is to plot speedup for the nine processes case *vs.* the ratio standard deviation/mean as was done in Figure 3.4. This very clearly shows the impact of the variance on the performance of the rootfinding procedure. Notice that when the coefficient of variation is much greater than one there can be no speedup obtained by incorporating multiple processes in the rootfinding task.

Now we compare the calculated no-overhead performance of the rootfinder to measured data observed on the machine. In order to measure performance as a function of the distribution of the $F(x)$ compute time a synthetic rootfinder was developed because we did not want to limit our investigations to particular distributions too early in the experiment. The nature of the calculation was still the real function evaluation, however the length of time spent computing was adjustable to reflect the distribution under consideration.

Figure 3.5 graphs performance in terms of elapsed time as a function of the number of processes for three distributions of the $F(x)$ calculation. In each case we compare theoretical performance to measured data. Because the means of the three distributions were not identical the data points for the single process instantiation do not coincide. Thus in this graph comparisons across distributions can only be relative approximations. What is important here is how close the measured curves are to their respective theoretical curves.

For each single process instantiation the measured and theoretical curves are far apart. This is because any pertubation the process experiences will be additive and will lengthen the basic cycle time.

As we incorporate more processes the constant distribution diverges the most from the theoretical while the exponential diverges the least. The reason for this behavior is that pertubations experienced by the processes will tend to increase the variance of the underlying distribution. However, a small change in the variance of the constant distribution will be a much larger relative change than a similar change to the exponential distribution.

That the observed data doesn't agree closely with the calculated curves is evidence that there are other influences on performance besides the distribution of the compute time.

3.3 The Variation in Performance of Individual Hardware Elements

The fluctuations in performance caused by the hardware will always be present because *Hydra* allocates C.mmp's resources dynamically. While a user cannot

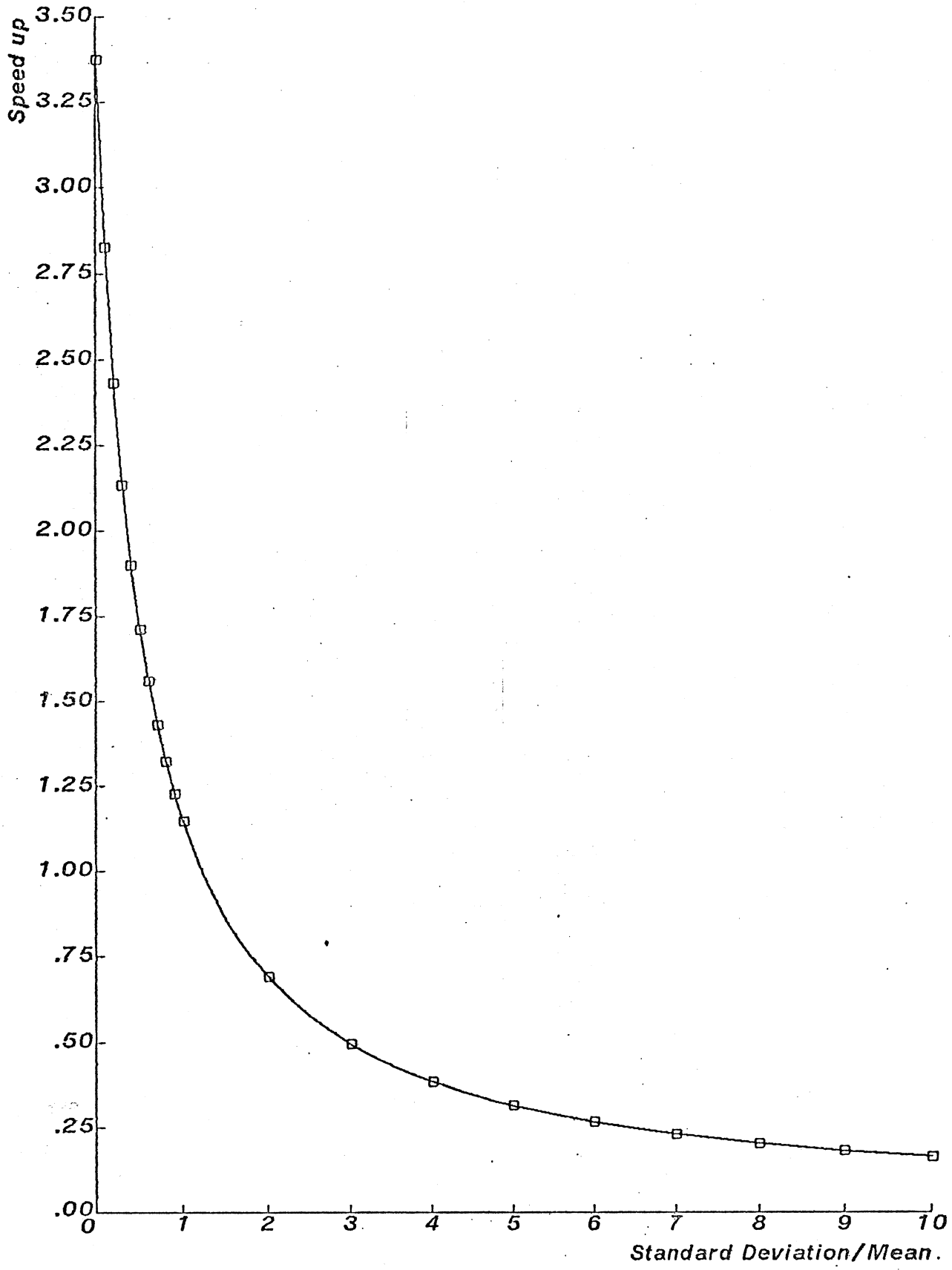


Figure 3.4 Speed Up vs. Coefficient of Variation for Nine Processes

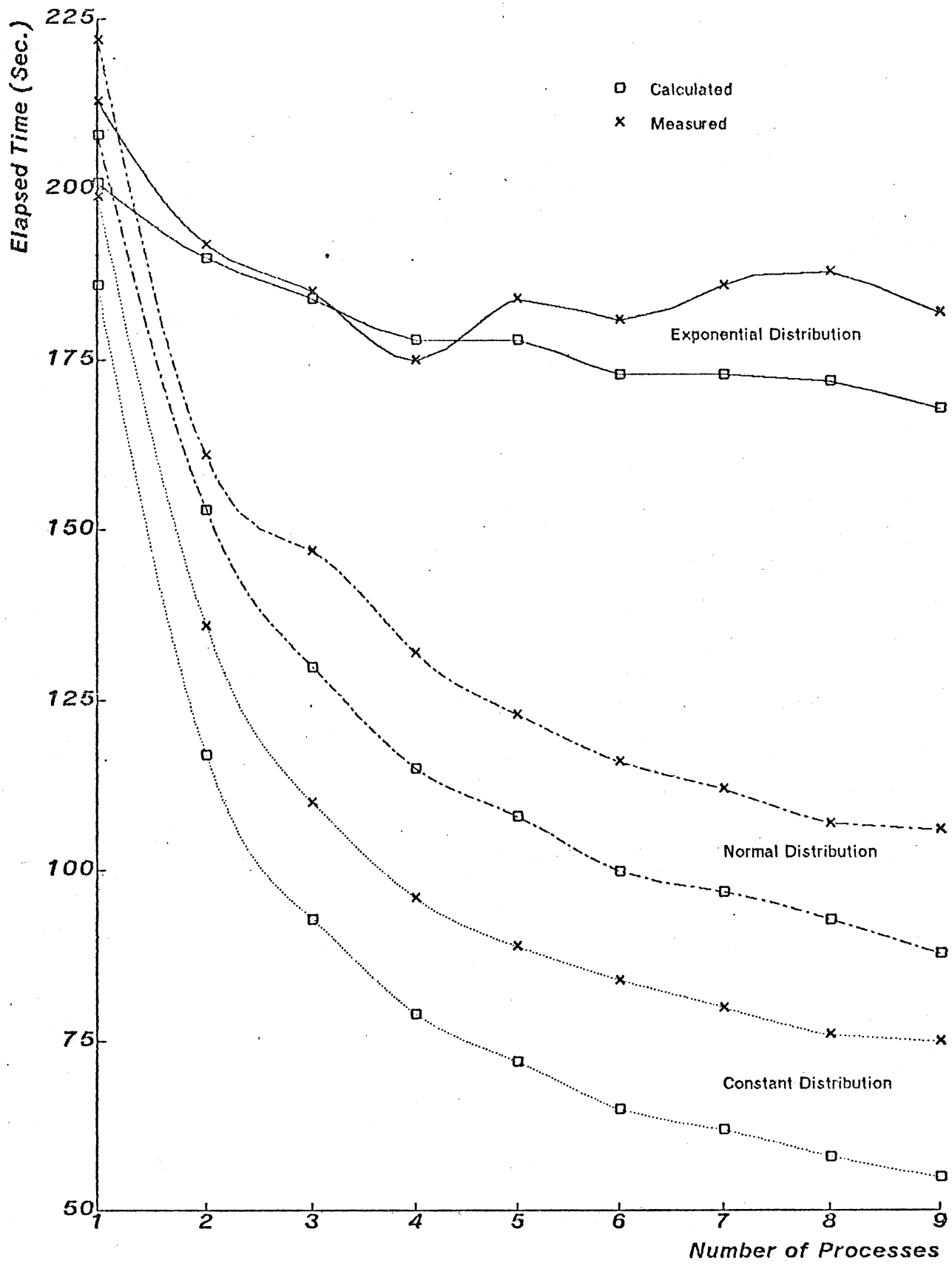


Figure 3.5 Measured Performance Compared to Calculated Performance

accurately predict the exact performance of his processes, he can estimate the magnitude of the fluctuation in performance by measuring the variation in the performance of the individual hardware elements.

3.3.1 Processor Related Variations

C.mmp is a multiprocessor constructed from PDP-11 model 40 and model 20 minicomputers. In Table 3.1 we have summarized the basic performance difference between the processors by comparing their execution of the $F(x)$ calculation without the presence of *Hydra*. Each processor performed the calculation 100 times in the same memory port. The number of *MSYN*'s[3] was counted and the elapsed time measured. These figures appear in the first and second columns. The third column of figures is the processor speed relative to Pc[0].

Pc	Model	Elapsed Time (sec.)	kMsyn's/sec	Relative to Pc[0]
0	11/20	15.559	443.3	1.000
1	11/40	10.413	662.4	1.494
2	11/40	9.985	690.8	1.558
3	11/40	9.745	707.8	1.596
4	11/20	16.144	427.2	0.963
5	11/40	10.060	685.7	1.546
6	11/40	10.238	673.7	1.519
7	11/40	9.829	701.8	1.582
8	11/40	14.867	463.9	1.046
9	11/40	10.022	688.3	1.552
10	11/40	10.173	678.0	1.529
11	11/40	10.001	689.7	1.555
12	11/40	10.129	681.0	1.536
13	11/40	10.005	689.4	1.555
14	11/20	14.965	460.9	1.039
15	11/20	14.999	459.9	1.037

Table 3.1 Speed Variations Among C.mmp's Processors

Naturally, a process on an 11/40 should execute faster than a similar process on an 11/20. Notice that even among processor of the same type there can be more than a 5% difference in speed.

Because there are two types of processors, the strategy of dynamically assigning processes to processors is complex. It is not sufficient to schedule a "ready" process to the best processor available. The following scenario clearly demonstrates why.

Suppose that the rootfinding processes are performing their function evaluations

[3] *MSYN* is the DEC name for the signal that indicates a request is being made for the Unibus™. Since only the processor is making requests the number of *MSYN*s is the number of memory requests made by the processor.

and are finishing at random times. After several have finished one would expect to find some idle 11/40's and computing 11/20's[4]. A good scheduler should handle its resources better. The idle 11/40's should "steal" the processes computing on the 11/20's. Naturally, this philosophy assumes that a *context swap* can be performed quickly. This process stealing philosophy is the scheduling policy on C.mmp.

3.3.2 Memory Related Variations

3.3.2.1 Technology Differences

C.mmp's centrally located primary memory is also a source of fluctuation in performance. The memory is divided into 16 modules, or banks. Each bank can service memory requests independently. However, the relative speeds of the banks are different because they contain different types of memory. At the time of this study 5 banks contained semiconductor memory and 11 contained magnetic cores. Table 3.2 summarizes the speed differences of the memory banks. In this experiment Pc[15] performed the $F(x)$ calculation 100 times in each memory bank. The elapsed times appear in the table.

Mp	Technology	Elapsed Time (sec.)	kMsyn's/sec	Relative to Mp[0]
0	core	15.243	452.5	1.000
1	core	14.943	461.6	1.020
2	core	15.127	456.0	1.007
3	core	14.999	459.9	1.016
4	core	15.087	457.2	1.010
5	semiconductor	15.950	432.4	0.955
6	core	15.272	451.6	0.998
7	core	15.402	447.8	0.989
8	semiconductor	15.887	434.2	0.959
9	semiconductor	15.858	434.9	0.961
10	semiconductor	15.860	434.9	0.961
11	semiconductor	15.855	435.0	0.961
12	core	15.070	457.7	1.011
13	core	15.155	455.1	1.005
14	core	15.034	458.8	1.013
15	core	15.013	459.4	1.015

Table 3.2 Speed Variation among C.mmp's Memory Banks

[4]During the course of our study the number of processors running in the system varied day to day. The processor configuration during the experiment with the synthetic rootfinder was 10 PDP-11/40's and 3 PDP-11/20's. Since we never used more than nine processors to perform the $F(x)$ calculation all of our processes ran exclusively on the 11/40's. However, the problem is real. If we could have incorporated more than ten processes into the rootfinding procedure we would have had to deal with it. Later experiments in this paper measure the impact of the non-homogenous processor configuration as the number of available 11/40's frequently was less than nine.

Notice again that even among memory banks of the same technology there are speed variations. These variations are small however, and are caused primarily by variations in the length of cable connecting a memory bank to the crosspoint switch and in the timing circuitry for individual memory modules.

3.3.2.2 Memory Bandwidth and Memory Interference

The previous experiments show the rates at which individual processors and memories can communicate. Another important characteristic is the maximum bandwidth of a memory bank. This rate will determine how many processors can compete for cycles in a single memory bank before the bank is saturated with requests. In this experiment all of the processors simultaneously executed the tight loop in the same memory bank. Two banks of different types were chosen to be representative of their respective technologies.

The results in Table 3.3 indicate that performance degradation will occur if more than two or three processors are competing for cycles in a memory bank. This implies that sharing code, a common practice to conserve memory space, will result in slower execution.

Semiconductor	$1489*10^3$	memory refs/sec.
Core	$1706*10^3$	memory refs/sec.

Table 3.3 Maximum Memory Bandwidth

In tables 3.4 through 3.6 we illustrate the performance degradation that results from sharing code. All of the measurements were performed on Pc[0]. In each case 100,000 total cycles were sampled. The first column, Memory Cycle Length, is the elapsed time from *MSYN* to *SSYN*[5], a complete memory cycle.

Table 3.4 is the control sample where we monitored the memory accesses while the system was idle. Although the vast majority of cycles were in the 500*ns.* to 1*us.* range there were some cycles that were greater than 14*us.* This is because a processor that doesn't have a process to execute runs a task called the *idle job.* The *idle job* consists of a *WAIT* instruction followed by the code that checks to see if there is a process to execute. This piece of code contains a critical section guarded by a mutual exclusion busy-wait loop. Since all of the processors are sharing this code and trying to gain exclusive access to the critical section there is a great deal of memory contention and the memory cycle lengths grow longer. We will use this table to compare the performance of the rootfinding processes when they execute from one common code page and when they each have a private code page.

Table 3.5 contains the results for when each of the processes executes from a private code page. Comparing this table to 3.4 we make two observations: while the average memory cycle length has increased slightly, there is relatively little difference

[5] *SSYN* is the DEC name for the signal that indicates the completion of a bus transfer. It is the signal the memory module uses to tell the processor that the memory access is completed.

between the two tables; the one category where a noticeable change does occur is the long (> 5.0 *us.*) cycles. There are less than half as many long cycles now because the processors are kept busy executing the rootfinding processes.

Compare these two tables to the results in table 3.6 where all of the processes share one common code page. Again we make two observations: the average memory cycle length has dramatically increased by 300%; more important still is that the percentage of long cycles (> 5.0 *us.*) has increased from .086% in table 3.4 to 15.6%, over two and one-half orders of magnitude more. This degradation in the basic cycle time will offset and eventually reverse speedup obtained by incorporating multiple processes in the rootfinding procedure.

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	65652	7787	14089	902
1.0 - 2.0	9470	1926	8	0
2.0 - 5.0	63	6	2	0
5.0 -14.0	63	6	10	0
14.0-50.0	5	2	0	0
> 50.0	0	0	0	0

Table 3.4 Memory Cycle Length Histogram for Idle System

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	65827	7461	11024	822
1.0 - 2.0	12705	1133	38	0
2.0 - 5.0	894	54	10	0
5.0 -14.0	28	3	0	0
14.0-50.0	1	0	0	0
> 50.0	0	0	0	0

Table 3.5 Memory Cycle Length Histogram with Private Code Pages

MEMORY CYCLE LENGTH	READ	READ-PAUSE	WRITE	WRITE-BYTE
0 - 0.5	0	0	0	0
0.5 - 1.0	52784	6504	9404	761
1.0 - 2.0	10810	689	102	0
2.0 - 5.0	3059	201	84	0
5.0 -14.0	14291	843	287	0
14.0-50.0	174	4	3	0
> 50.0	0	0	0	0

Table 3.6 Memory Cycle Length Histogram with Common Shared Code Page

Figure 3.6 captures the impact of the finite memory bandwidth problem on the rootfinding procedure. We have graphed the elapsed time to locate 50 roots versus the number of processes for two implementations of the rootfinding procedure. The dashed curve results when a single copy of the code page is shared. The solid curve

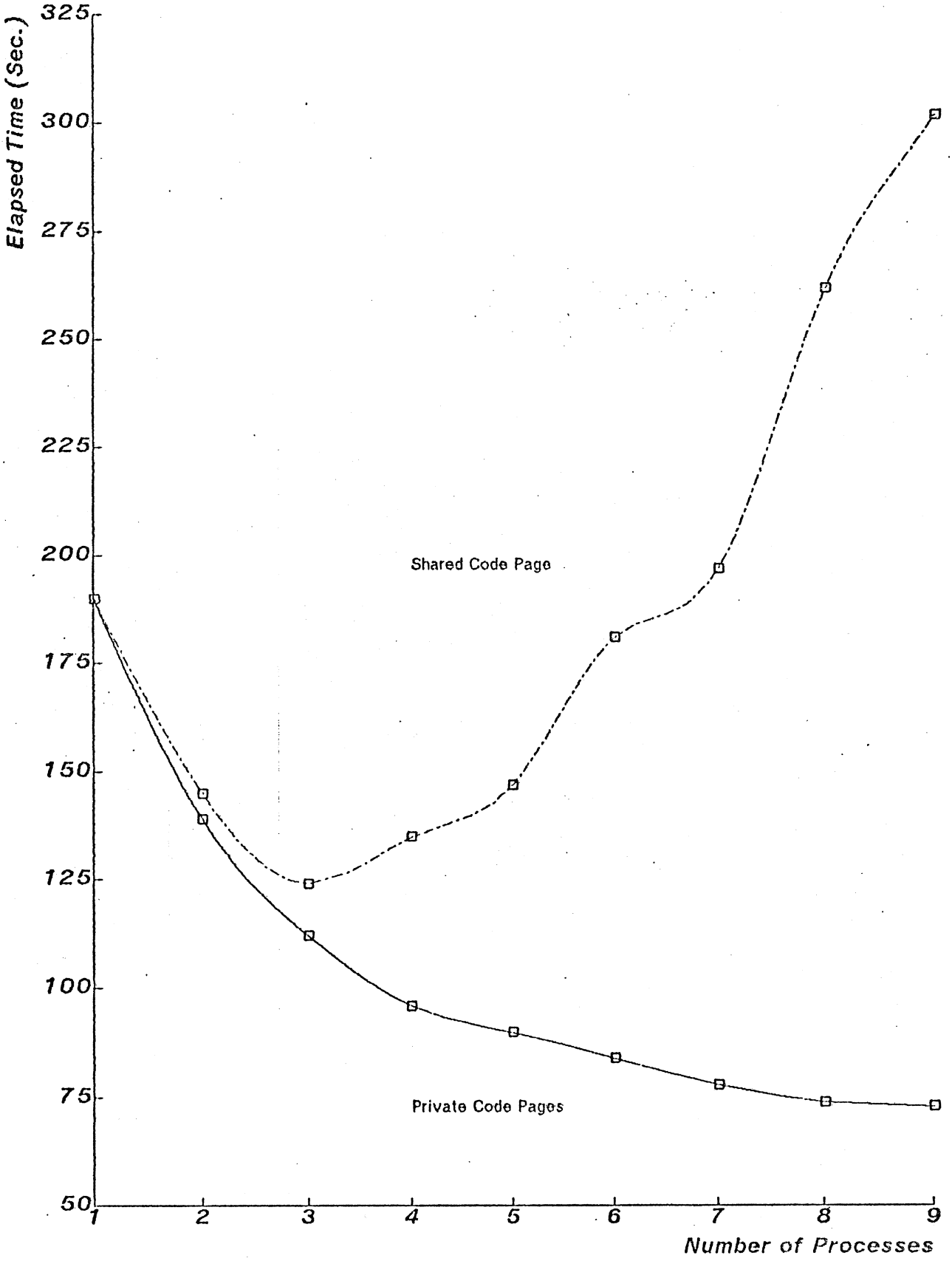


Figure 3.6 Performance Degradation Due to Finite Memory Bandwidth

is the performance when the cooperating processes each have a copy of the code in a private memory bank.

This graph also can provide some insight into the speed versus space tradeoff. If we compare the speedup over the single process instantiation for both the shared and no-sharing versions of the rootfinder we find that the no-sharing version has a maximum speedup of 2.60 using nine processes while the shared version's performance peaks at 1.53 using three processes. Neglecting the single global data page we have a achieved a 170% increase in speed at the cost of a 300% increase in size. In this study memory is plentiful and we squander space for speed.

One obvious solution to the speed *vs.* size tradeoff is to interleave the central memory on the low order bits rather than the high order bits. This solution would tend to scatter memory requests more evenly across the 16 banks. To maintain availability it might be necessary to organize the store as four banks of 4-way interleaved memory. A second solution is to give each processor a cache to work with. This is the solution currently being implemented on C.mmp.

3.4 Operating System Related Performance Fluctuations

3.4.1 Introduction

The operating system (*Hydra*) also perturbs the performance of the rootfinding procedure. Although C.mmp was intended to be a multi-user multi-programming facility, it is possible to use the machine in a dedicated single user mode. In this mode of operation the user can for the most part minimize any interference from *Hydra* by simply not doing anything that requires operating system assistance. Most of the measurements in this study were performed in this way. However, certain functions (i.e. scheduling of processes and I/O interrupts) must be performed by *Hydra* and cannot be avoided. The contribution to performance fluctuation from these basic operating system functions is measured and discussed in the following sections.

3.4.2 The Kernel Tracer

The *Tracer* is a software monitor that can obtain information about significant activity on C.mmp under the *Hydra* Operating System. Since it is a software monitor, the *Tracer* does perturb the timing data it attempts to measure, however this can be compensated for in the post-processor software.

The *Tracer* can monitor such things as: context swaps (this occurs when a processor changes from one process to running another), semaphore activity, process starts and stops, O.S. requests (*Kernel Kalls*) and a multitude of other events. Events defined by user programs may also be traced.

The data is collected in real time and later post-processed offline. There are

numerous post-processing programs that produce various forms of output (by process or processor dumps, time-line execution histories, and various statistical analysis packages).

All of the *Tracer* data that follows is in the form of a processor time-line execution history. We use various symbols in the trace to encode events in order to compact the data. Table 3.7 contains these symbols and their meanings. Each row of the trace represents the activity on a processor. The time in seconds appears along the bottom edge. We will discuss in detail the first trace which captures the impact of I/O interrupts on performance.

3.4.3 I/O Devices and Interrupts

Random interrupts from I/O devices and processors contibute to performance fluctuations in the rootfinder processes. Unlike the memory, I/O devices are not centrally located and accessable through an *nxm* crosspoint switch. Devices are associated with a particular processor. Thus, for example, a read or write from a disk on Pc[0]'s Unibus must be performed by processor 0 regardless of which processor initiated the request. Since interrupts are serviced by stealing cycles from the currently executing process large fluctuations in compute times can be found for processes running on processors with I/O devices.

In Figure 3.7 interrupts associated with I/O perturb the performance of the rootfinding processes. C.mmp's processor configuration during this trace was Pc[0, 3, 4, 5, 6, 7, 8, 9, 11, 12, and 13]; and appear from bottom to top as rows of the trace. Pc[0, 4, and 8] are PDP-11/20s and the rest are PDP-11/40s. Processes(35, 43-50) are the nine rootfinding processes. Process 29 and the DAEMON are other processes that happened to be awake at the time. These two "other" processes are doing things that cause a substantial amount of I/O. The following discussion describes how this I/O activity perturbs the rootfinding processes.

A previous iteration finishes at 0.612 seconds into the trace. Process 50, P(50), on Pc[11] was the last to finish its calculation (the activity on Pc[6] is P(29)) and begins to wake its sleeping companions by unlocking their semaphores. One by one the processes wake up and begin to perform the next iteration. P(50) finishes waking up all the processes (P(49) was the last to wake up at .641) and begins its own function evaluation. One by one the processes finish their calculations and post their results, after which they "*P*" their semaphores and wait for the beginning of the next iteration. When they block on the semaphore they are removed from the processor (e.g. CSW for P(45) on Pc[5] at .700). Notice that four of the processors have large chunks of time shaded between brackets. This denotes an interrupt service routine performing I/O to a device on that Pc's Unibus. Interrupt service routines can consume between 1 and 15 milliseconds of time. This causes the rootfinding process on that Pc to arrive at the synchronization point late, thus lengthening the STAGE time.

For example, P(49) on Pc[8] is interrupted at .681 for 13 milliseconds and then again at .707 for 4 more milliseconds. Notice however, that P(49) on Pc[8] switches to Pc[6] at .709 and finishes its function evaluation at .728 uninterrupted. Since it is the

Table 3.7 Tracer Symbols

PROCESS N	: PROCESS #N IS RUNNING
- CSW -	: A CONTEXT SWAP
IOT #X	: SPECIAL TYPE OF KERNEL KALL
KALL #X	: KERNEL KALL #X
RET X	: RETURN VALUE FROM A KERNEL KALL
n [	: START OF AN INTERRUPT AT LEVEL N
I	: INTERRUPT SERVICE ROUTINE EXECUTION
]	: END OF AN INTERRUPT
EVENT X	: USER DEFINED EVENT X OCCURS
P	: P OPERATION ON A SEMAPHORE
V	: V OPERATION ON A SEMAPHORE
DAEMON	: OPERATING SYSTEM PROCESS
¦¦¦¦¦¦	: IDLE TIME

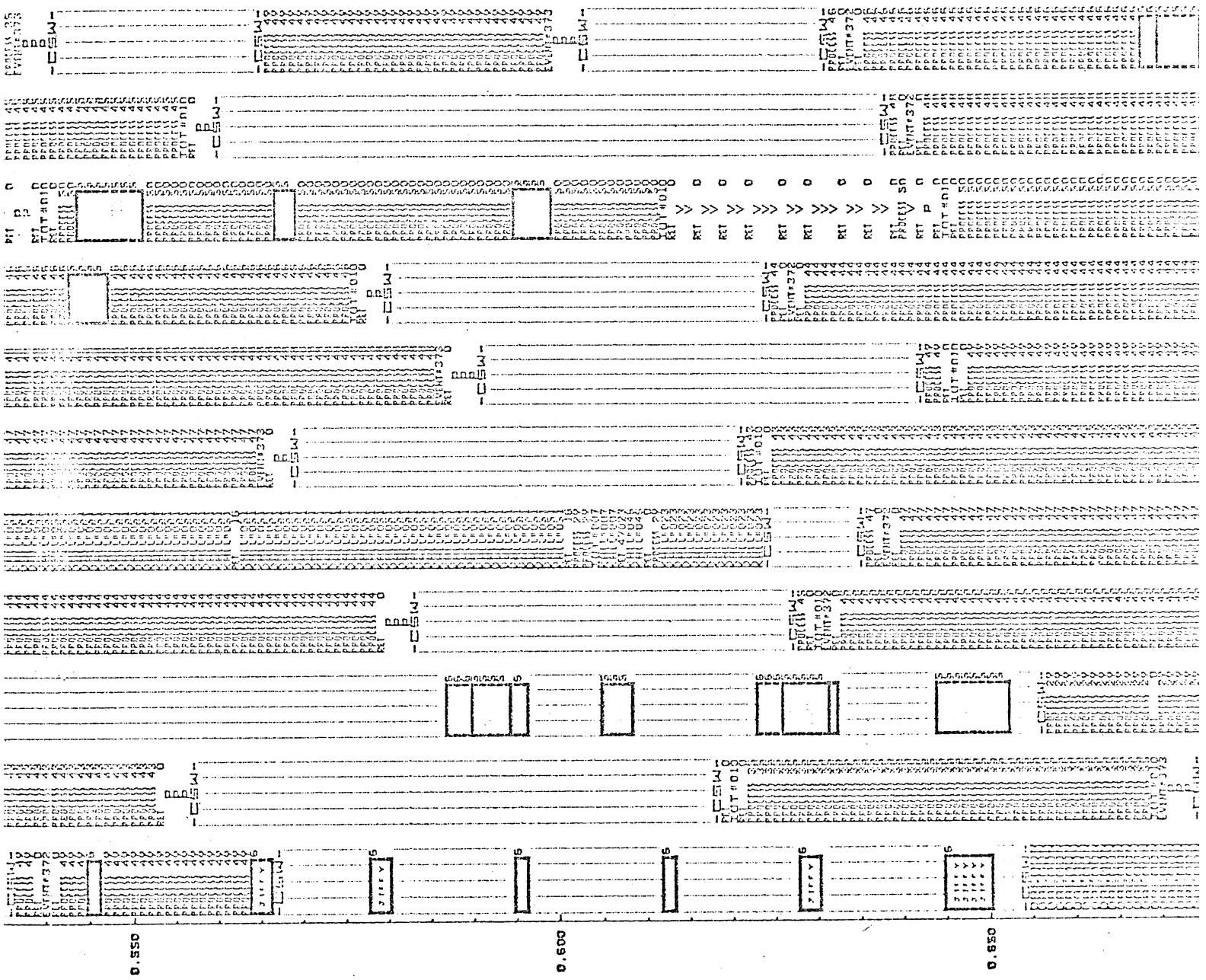


Figure 3.7a Pertubations from Interrupts

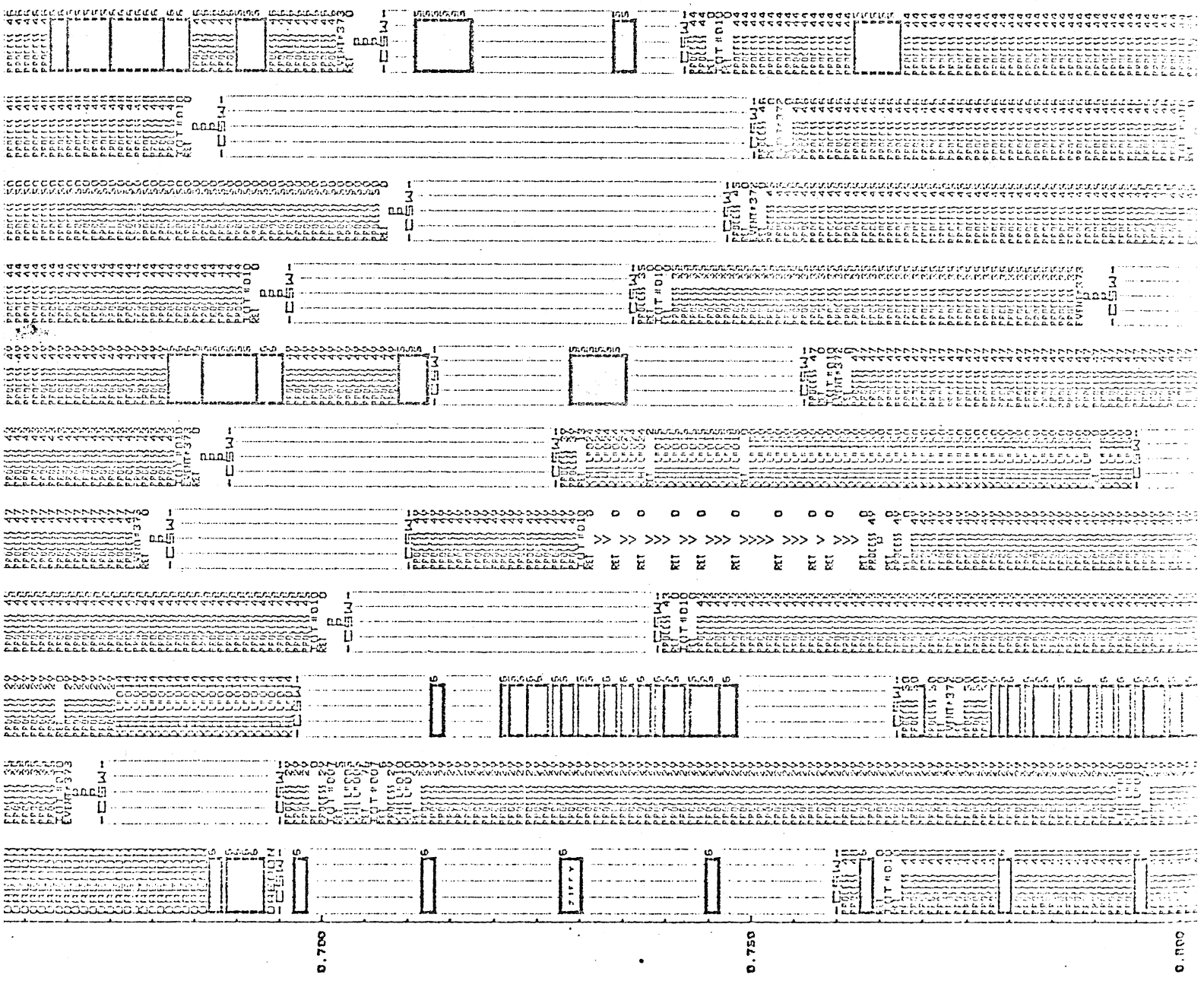


Figure 3.7b Pertubations from Interrupts

last process to finish it assumes the jobs of finding the new root containing subinterval and dispatching the processes to perform the next iteration.

In this example the interrupted process was delayed enough to become the last process to finish thus lengthening the STAGE time. This is not always the case. For example, P(46) on Pc[13] was also interrupted during its function evaluation for a approximately 21 milliseconds yet it was not the last to finish and did not cause the STAGE time to lengthen. This is another advantage the multiprocess implementation of the rootfinding procedure has over its uniprocess counterpart. In the single process instantiation the interrupt time is additive and each occurance lengthens the iteration. In the multiprocess version only the interrupt time associated with the last process to finish is additive.

3.4.4 Kernel Processes and Special Functions

Operating system requests are frequently handled by special high priority *Kernel* processes and as such perturb the cooperating rootfinder processes by stealing processors. Of particular interest are the processes that perform scheduling. Because synchronization of communicating processes can involve rescheduling the processes the special scheduler processes can become bottlenecks causing performance degradations.

During the trace of Figure 3.8, C.mmp's processor configuration was Pc[0, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, and 13]. Of these, 4 and 8 are 11/20's (so is Pc[0]) and are the 3rd and 7th blank columns with no execution history. This is because there were enough processors of the prefered (11/40) type so that the 11/20's were never used. Similarly Pc[12] was not needed.

In this trace processes (18, 19, 20, 21, 22) are rootfinding processes. Processes 1 and 2 are *Kernel* scheduling processes, and process 14 is the *Tracer* process.

P(22) on Pc[10], the last process to finish the previous function evaluation, initializes the necessary parameters for the next iteration. At 285 ms. into the trace (.285) it begins to "*V*" its "sleeping" companion processes, and at .309 it begins its own function evaluation (event 372).

Meanwhile P(2) on Pc[6] (scheduling process) wakes up CSW at .293 and begins to perform the task of actually waking up the processes process 22 has just "*V-ed*". It is a relatively painfull task involving several semaphore operations and several *Kernel Kalls* per process. Finally process 18 (the first to be "*V-ed*") wakes up and begins its function evaluation at .348, approximately 60 ms. after process 22 performed the *V* operation.

To expedite the costly wake up procedure processes 1 and 2 (scheduling processes) cooperate to start and stop the rootfinding processes. Unfortunately by the time they get around to starting process 21 (the last process that is to wake up), 3 of the other rootfinding processes have already finished their function evaluations and have gone back to sleep (*P* followed by CSW). A full 130 ms. have transpired since process 22 performed the *V* to wake process 21!

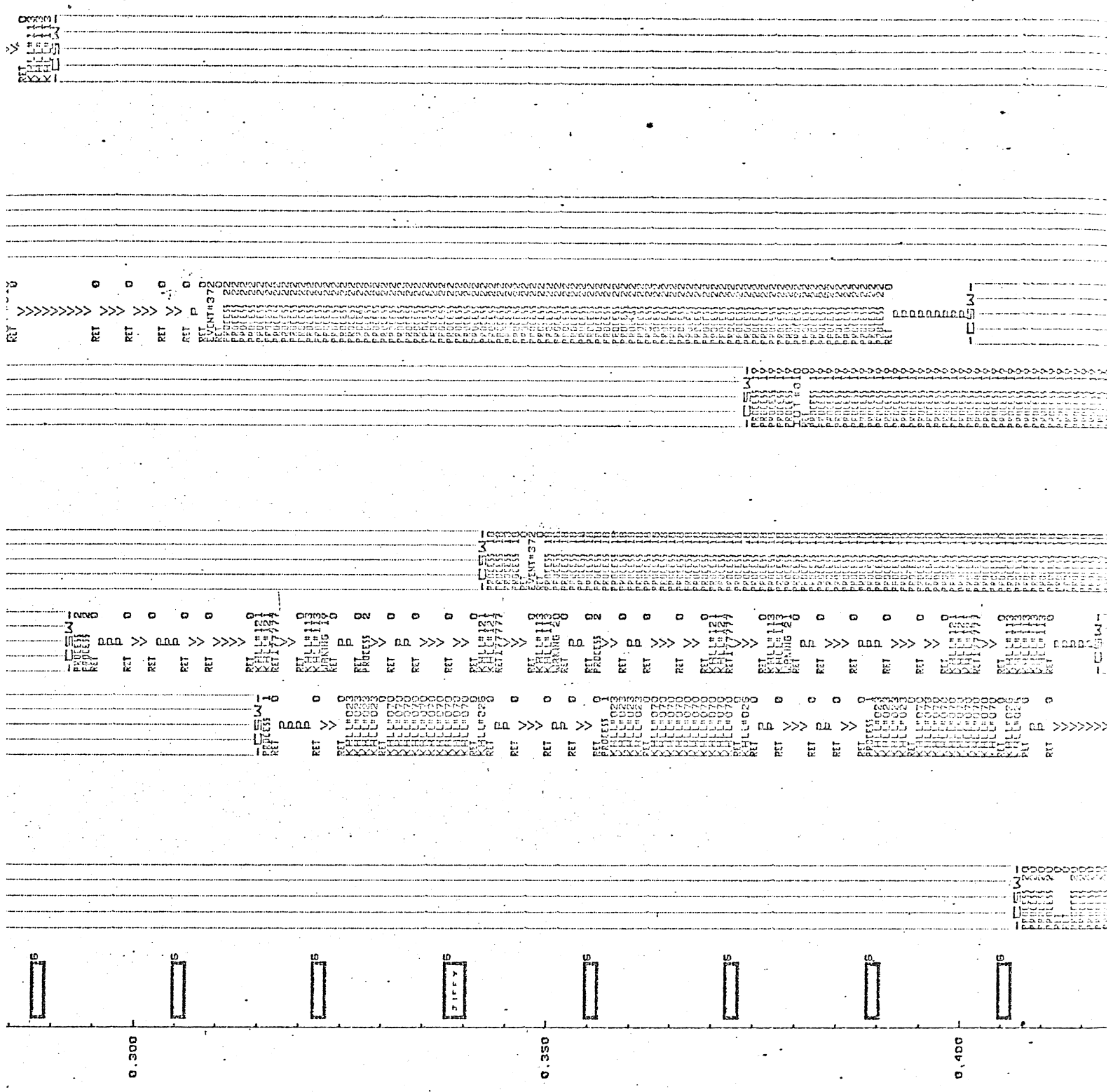


Figure 3.8a Pertubations Induced by Operating System Processes

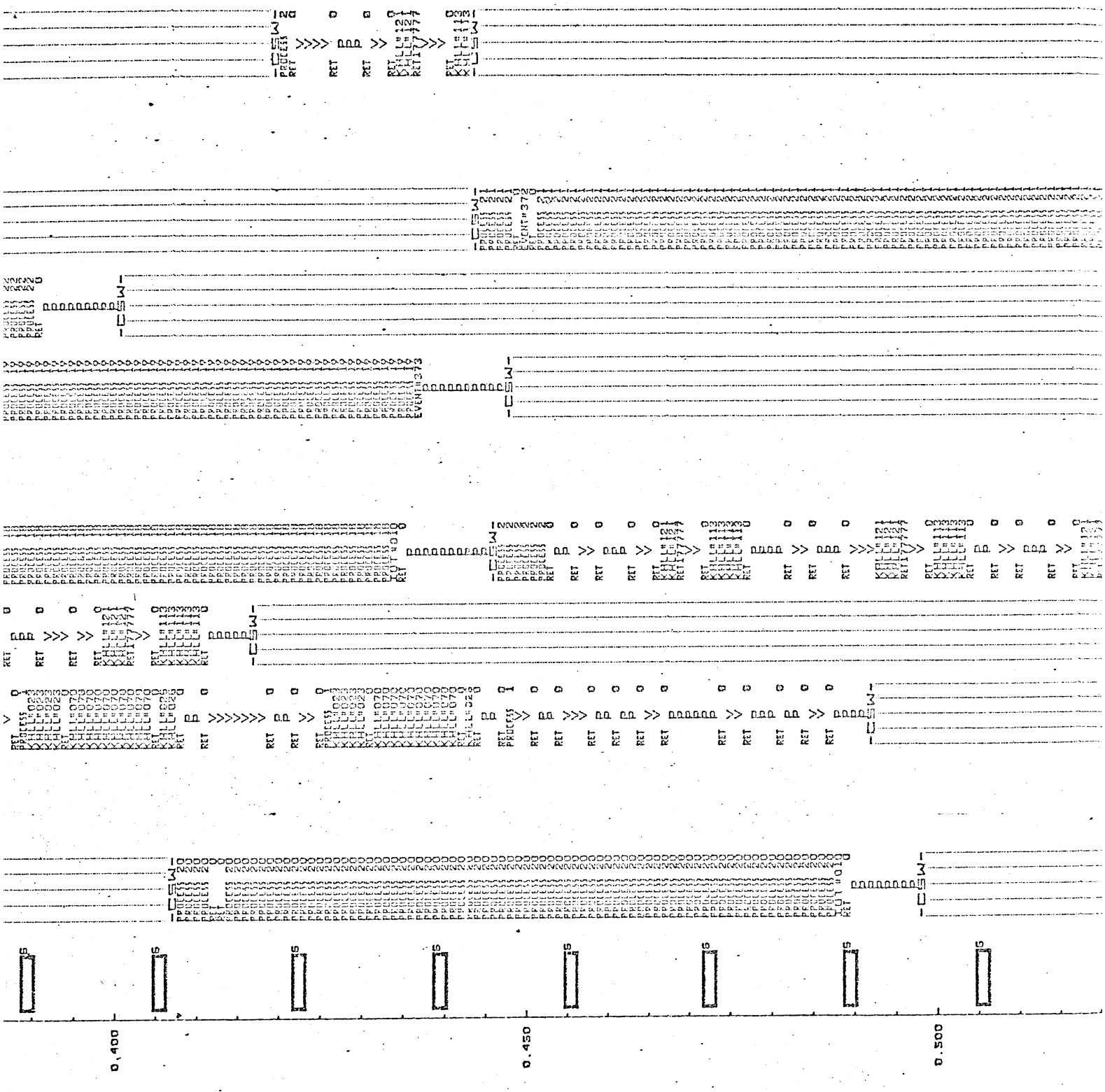


Figure 3.8b Pertubations Induced by Operating System Processes

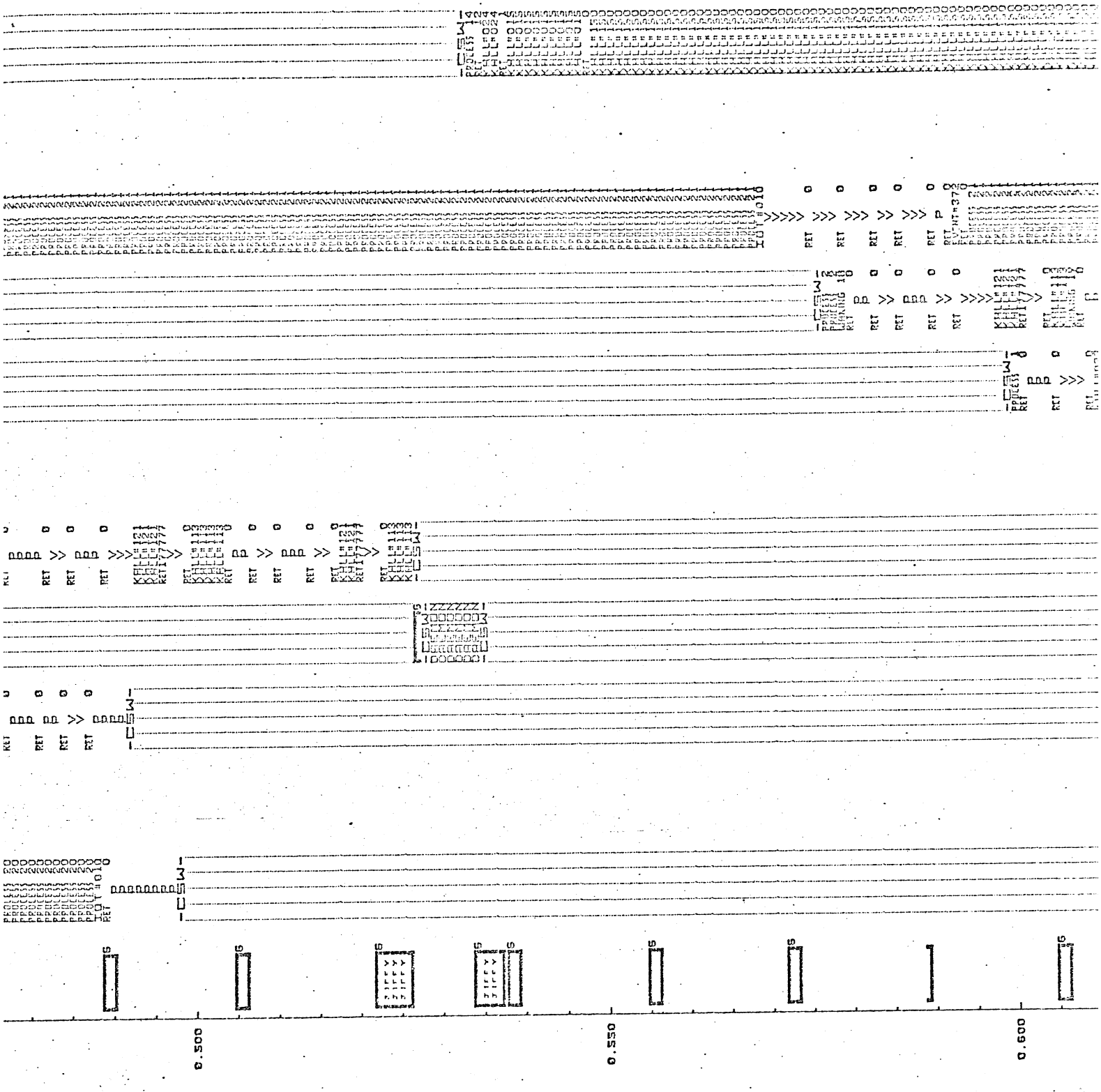


Figure 3.8c Pertubations Induced by Operating System Processes

Another side-effect related to the O.S. that can effect the performance of cooperating processes is the round-robin scheduling of processes under *Hydra*. This traditional policy is implemented using the notion of "time-sliced" intervals of execution to provide equal service to all tasks. Occasionally a process exhausts its time slices and must ask for more. This can take a considerable amount of time (greater than 150 msec.) Whether or not the time-slice end anomaly will perturb the performance of the cooperating processes depends upon the average duration of the function evaluation and the frequency of the time-slice end condition. In this study a process must consume 10 one half second slices before encountering the time-slice end condition.

Figure 3.9 is the distribution of the elapsed time to perform an $F(x)$ calculation in the presence of *Hydra*. The long tail in the distribution is a result of the time-slice end condition occurring for the process performing the function evaluation. Compare this histogram to the one in Figure 3.1.

3.5 Summary

We can integrate all of the performance fluctuations into a single measure by running the rootfinding procedure several times without changing any of the user controlable variables. By keeping all of these parameters (e.g. number of processes, function evaluation distribution and mean) constant we can measure the combined performance fluctuation from the hardware and operating system.

Figure 3.10 is the result of this experiment. Each sample point represents the time that elapsed while 9 cooperating processes executed the rootfinding procedure 50 times.

Although many more samples are needed to establish an accurate measure of this performance fluctuation it is already apparent that many samples are necessary just to accurately plot a single point on a performance curve. For all of the performance curves in this paper we have been forced to run the experiment for each data point several times. We then plot the fastest time as the representative value because it will be the one with the fewest random pertubations.

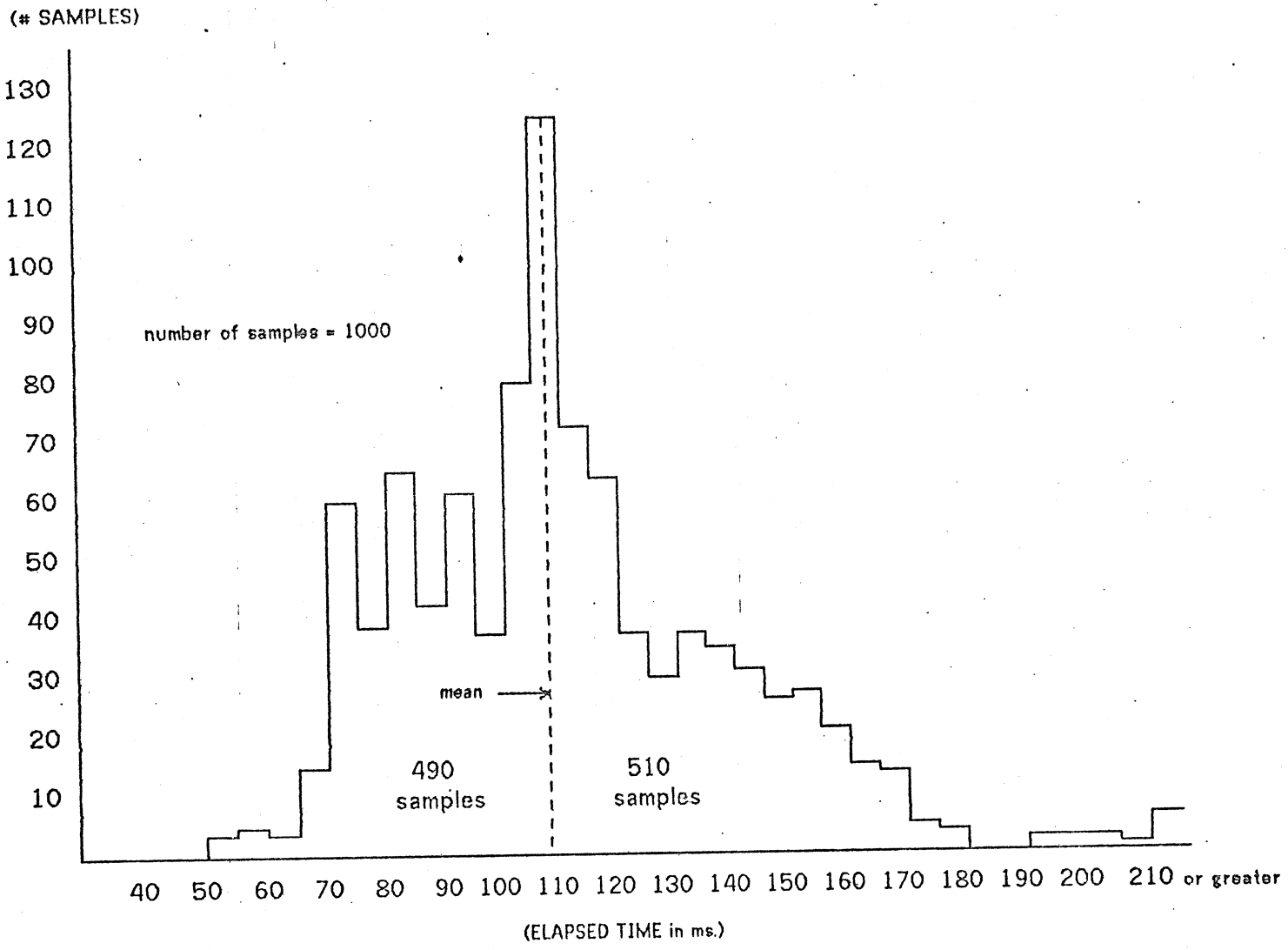


Figure 3.9 Distribution of the Time to Calculate F(x) in the Presence of HYDRA

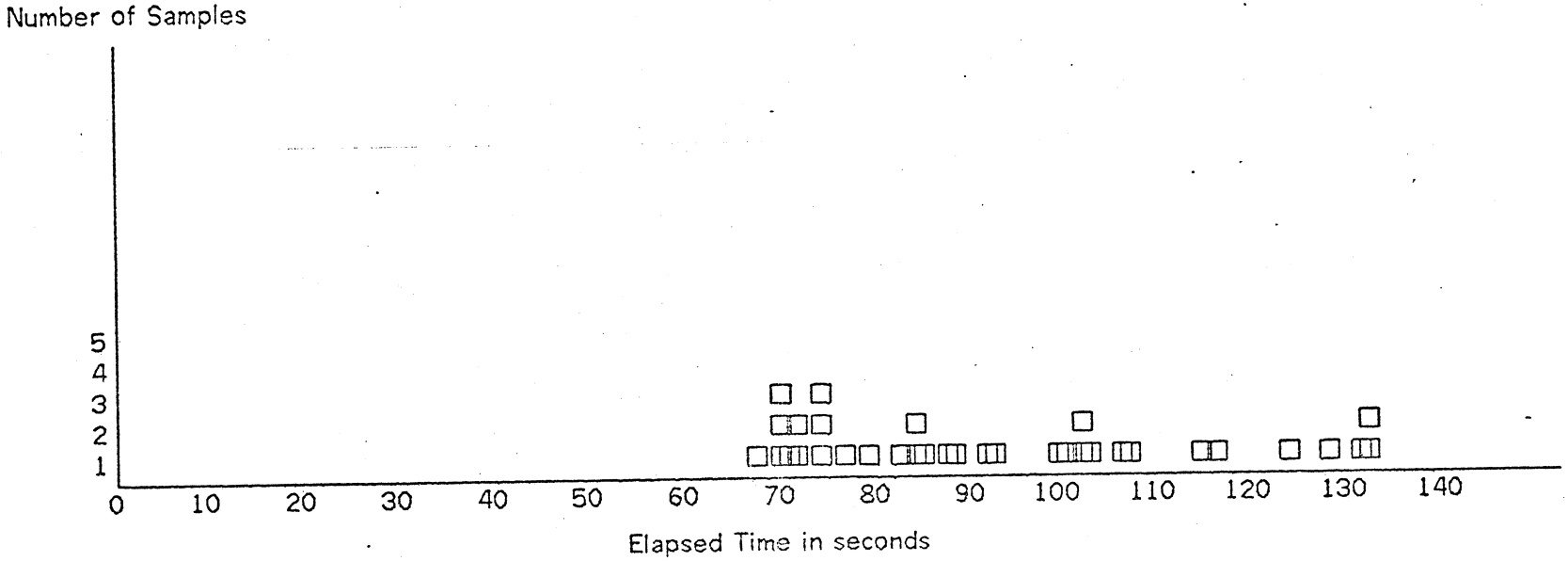


Figure 3.10 A Summary Graph of Performance Fluctuations

4. The Effect of Sychronization on Performance

4.1 Introduction

As synchronization of cooperating processes is a fundamental problem in the implementation of a parallel algorithm we will discuss the tradeoffs of the various synchronization mechanisms available to the C.mmp user.

Up till now we have used a very simple form of "busy-waiting" loop to synchronize the cooperating processes. Although synchronization using this method is extremely fast there are also undesirable side effects that can cause serious performance problems. We will begin by discussing several alternative synchronization mechanisms, describing there functionality and any interesting side effects, comparing their performance in the context of the rootfinding algorithm, and conclude by presenting the range of usefullness for each.

4.2 Description of Synchronization Primitives

Let us first examine the nature of the synchronization problem for the rootfinding processes. In figure 4.1 we present a more detailed view of the STAGE time and in particular focus on the mechanics of synchronization. The segment labeled FIND is the time spent locating the new root containing sub-interval. The *V(i)'s* correspond to waking up each of the rootfinding processes. One quickly notices that the overhead of synchronization can be a significant part of the STAGE time in certain instances. Because we have used a *spin lock* (a form of busy waiting) to synchronize the processes thus far, the overhead of synchronization has been negligable. Unfortunately it is not always desirable to implement synchronization with this mechanism.

4.2.1 The Spin Lock

Of the three synchronization primitives considered in this study, the spin lock is the most rudimentary. This primitive is actually implemented independent of any *Hydra* support and is nothing, more than a tight loop in which the process continually tests a semaphore until it can set it successfully. In fact, the *P* and *V* operations are the following PDP-11 code sequences:

```
P:  CMP SEMAPHORE, #1     ;SEMAPHORE=1?
    BNE P                 ;loop until it is 1
    DEC SEMAPHORE         ;Decrement SEMAPHORE
    BNE P                 ;If neq 0 go to P

V:  MOV #1, SEMAPHORE     ;Reset SEMAPHORE to 1
```

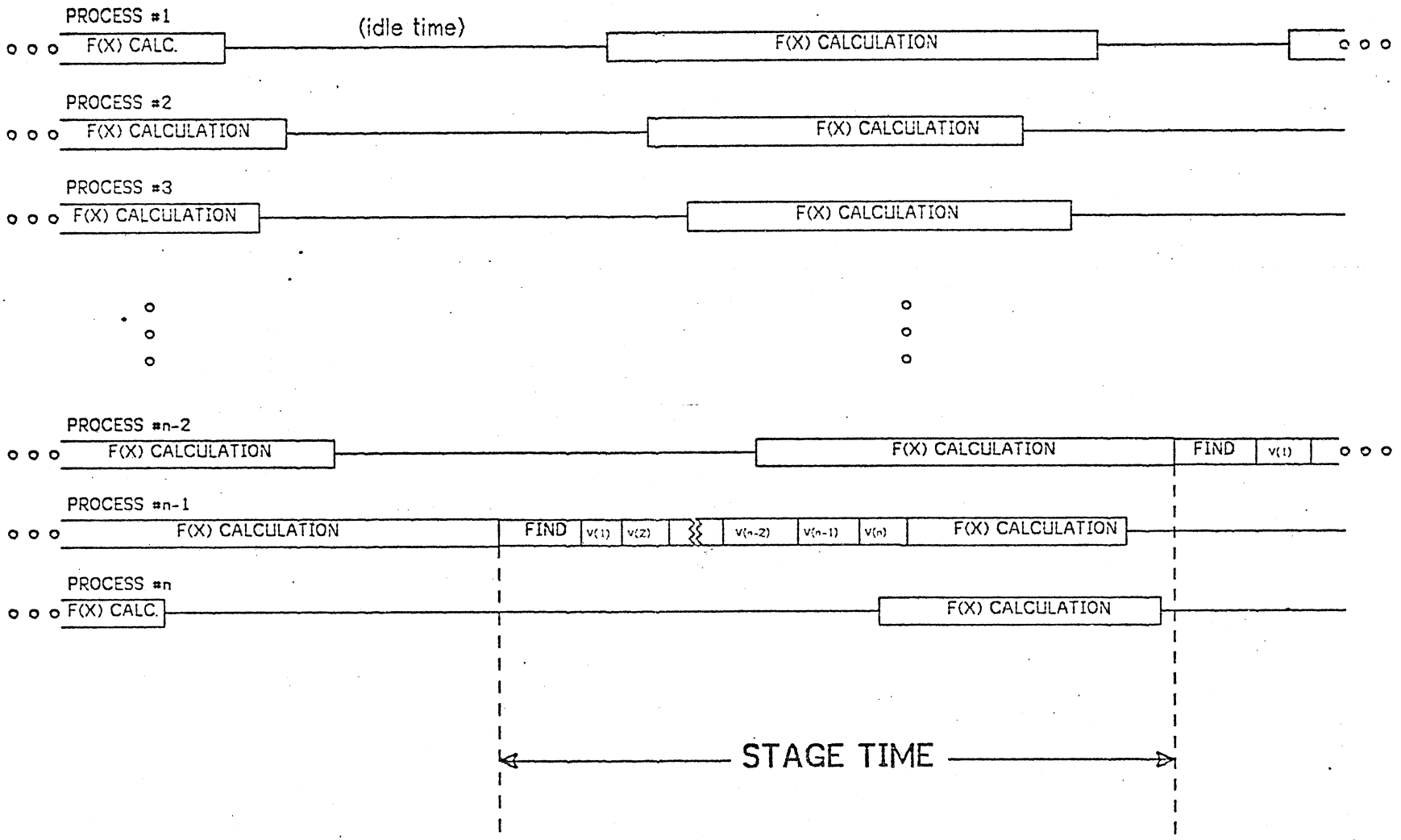


Figure 4.1 A Detailed View of the STAGE Time

The repeated polling of the semaphore although extremely fast, has two very undesireable qualities.

The first is that when the process completes its function evaluation and starts to poll the semaphore while waiting for its counterparts for finish, the processor is not free to perform useful work.

The second major drawback is that the polling process is comsuming many cycles in the memory bank that contains the semaphore. As more process finish their function evaluations and begin to poll the semaphore the bandwidth of the memory bank is quickly consumed. The unfortunate process that has its code page located in the bank with the semaphore will be competing for cycles with many "busy" processors. This second problem can be circumvented somewhat by inserting a tiny delay loop in the semaphore code (i.e., decrement a register to zero before checking the semaphore). This will decrease the frequency of memory requests in the semaphore memory bank, while not slowing the sychronization primitive appreciably. However, the primary problem still remains; a "spinning" process prevents a processor from doing more useful work.

4.2.2 The Kernel Semaphore

The *Kernel* semaphore (K-SEM) is implemented by the *Hydra* operating system. It is the low level synchronization mehcanism used by system processes. When a process blocks or is to wake up, a state change for that process is made inside the *Kernel*. Because it is implemented within the domain of the *Kernel* the user evokes operations on the semaphore (*P* and *V*) by issuing *Kernel Kalls*. If the process blocks while trying to *P* the semaphore the *Kernel* swaps the process from the processor and places the process in the semaphore's blocked-queue, where it will remain until another process *V*'s the semaphore. When the process can proceed again it is swapped back onto an available processor and continues execution from where it was blocked. The important attributes of the *Kernel* semaphore are:

1. A blocked process is swapped from a processor.
2. When a process blocks its pages are kept in primary memory. This ensures that the process will quickly resume execution when it is swapped back onto a processor.
3. The *Kernel* semaphore is approximately two orders of magnitude slower than the spin lock.

4.2.3 The Policy Module Semaphore

The policy module semaphore (P-SEM) is implemented by the scheduling subsystem called the *Policy Module* (PM). This primitive is intended as the user's primary mechanism for performing synchronization.

Because the synchronization is performed within the context of a system

process there is much more flexibility in how a blocking/waking process is handled. The first policy that was adopted to handle blocking/waking processes was the following:

1. There would be 2 PM processes that would cooperate to perform synchronization operations for users; one would start and stop processes and the other would handle communication between the *Kernel* and user.

2. When a process blocked on a semaphore it would be context swapped from the processor.

3. Any 'dirty' pages belonging to the process would be updated on secondary storage.

4. When a process was to wake up it would be restarted by one of the PM processes after all the swapped out pages belonging to the process were brought back in to central memory.

This policy has evolved into a much faster arrangement of multiple processes in the current version of the PM.

One modification to the PM that was found to improve performance substantially was to delay the updating of a process' dirty pages onto secondary storage. Often a process is blocked for very short amounts of time and will quickly resume execution after only several milliseconds of waiting for a certain condition to be true. However, when a page is to be updated onto secondary storage it is written onto one of several IMS^{TM} fixed head disks which will take at least 32 milliseconds per page. The swapping disks revolve once every 16.67 milliseconds and it takes two revolutions to update a page: one to write it out and the second to perform a read-check operation to validate the copy. Thus it is quite possible for a process to spend most of its time blocking and unblocking if the inter-synchronization interval is small enough. The problem would be even more severe if there were a task force of cooperating processes (e.g. the rootfinding processes) blocking and unblocking every few milliseconds.

The current version of the PM initializes the delay time parameter, (ϵ), to 300 milliseconds. Table 4.1 is a summary of the time it takes to perform the basic semaphore operations on the various primitives.

	Spin Lock	K-SEM	PM0	PM1(ϵ=0)	PM1(ϵ=300)
Time for a process to do a *V* (*us.*)	30	3000	6000	5000	5000
Time till a process wakes up from a *V* (*us.*)	30	5000	55000	50000	13000
Time from *P* to CSW (*us.*)	na	3000	9000	6000	6000
Time spent in PM while waking a process (*us.*)	na	na	62000	20000	0

Table 4.1 Comparison of Execution Times for Semaphore Primitive Operations

4.3 The Impact of Synchronization on Performance

4.3.1 Introduction

Now that we have described the functionality and presented the individual performance statistics for the basic primitive operations we can observe the impact of synchronization on the performance of the rootfinder. Up till now we have eliminated most of the overheads associated with synchronization by using the spin lock primitive. The remainder of the paper examines the rootfinder's performance as we employ the alternative synchronization primitives.

4.3.2 Comparison of Primitives When Compute Time ~ Synchronization Time

The first graph, Figure 4.2, compares the performance of the various implementations of the rootfinder using different primitives to perform the process synchronization. We have plotted the elapsed time to find fifty roots as a function of the number of processes. This data was generated by the authentic (not synthetic) roootfinder. The distribution of the $F(x)$ computation being approximately Normal with mean 72 milliseconds and standard deviation 18 milliseconds[6]. We compare the performance of four alternative synchronization primitives: spin lock, K-SEM, PM1(ϵ=300), and PM0 semaphores.

The curve for the PM0 semaphore implementation exhibits degradation as we increase parallelism. The reason for this behavior is that the overhead of synchronization is greater than the average compute time. In other words, a process spends more time synchronizing than computing. In this instance we would be better off using a single process.

[6]On and 11/40 processor

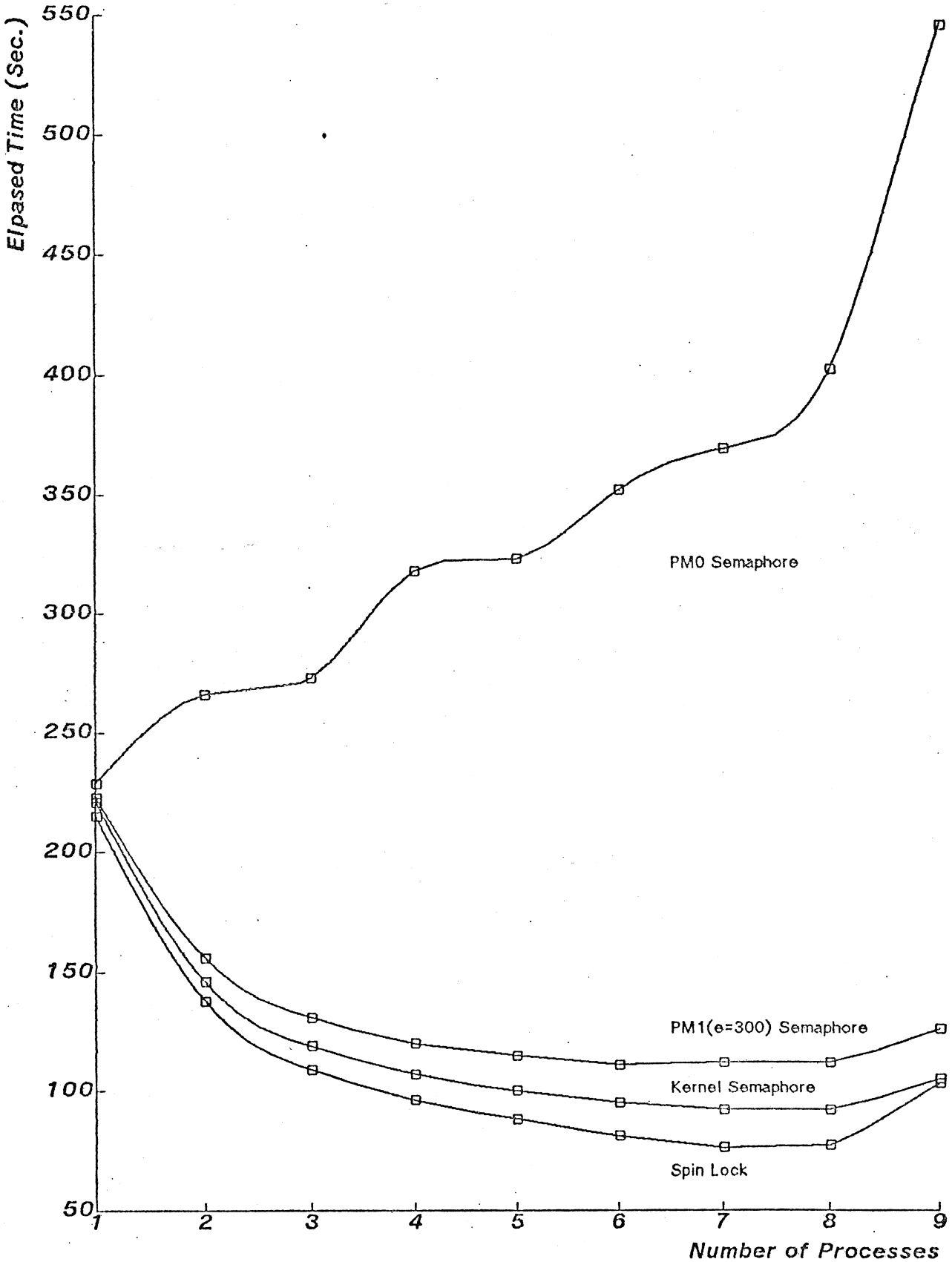


Figure 4.2 A Performance Comparison of Synchronization Primitives

The curve for the PM1(<=300) semaphore implementation depicts substantially better performance than its predecessor. Performance reaches a maximum speedup of 2.00 at six processes. There is no additional speedup gained by employing more processes and in fact there is a noticeable degradation at nine processes. This sudden degradation occurs because of the non-homogenous processor configuration (NHPC). During this experiment C.mmp's processor configuration was eight 11/40's and one 11/20. Thus when we incorporated the ninth process it ran on the slower 11/20 type processor causing the STAGE time to lengthen yielding an overall slower performance.

The K-SEM implementation has its peak performance of 2.4 at eight processes. It too is affected by the NHPC problem and performance degrades slightly at nine processes. The overall performnce of the K-SEM implementation is about midway between the PM1(<=300) and the spin lock versions.

The spin lock implementation has by far the best speed up maximum of about 2.8 for eight processes. One interesting point is that the NHPC problem causes a much more severe performance degradation for this semaphore than for the others[7]. The reason is that the processes blocked on the spin lock semaphore remain on their processors, whereas the other implementations free the faster 11/40 type processors to steal the process that is still running on the slower 11/20 processor.

4.3.3 Comparison of Primitives when Compute Time >> Synchronization Time

In the previous experiment the overhead of synchronization was in some cases a considerable fraction of the STAGE time. If we make the compute time for the function evaluation much larger thus reducing the percentage of time spent synchronizing, the performance differences between the various implementations is also reduced. Figure 4.3 graphs performance in terms of speed up as a function of the number of processes. We used the synthetic rootfinder again to generate $F(x)$ computations that take 375 milliseconds to compute, the distribution being a constant. The dashed curve is the prformance obtained using the PM0 semaphore and the solid curve that obtained using the spin lock.

We expected the curves to be closer together yet the spin lock version outperforms the PM0 semaphore 2.8 to 2.1 at maximum speed up. The reason for the large difference is that the PM processes must perform the semaphore operations *seriolly*, each V operation taking about fifty-five milliseconds. Thus the n^{th} rootfinder process is not started until 55*n milliseconds into the STAGE time. In this manner the ninth rootfinder process does not complete its function evaluation until 870 milliseconds have past. Similarly, when the rootfinder processes complete their $F(x)$ calculations the PM processes again *seriolly* perform the P operations on the semaphores causing still further performance degradations.

[7]The PM0 implementation performance curve has a greater degradation than the spin lock version however the reason is not merely the NHPC problem. The primary reason is instead that the two PM processes that perform the semaphore operations are almost constantly running, consuming processors.

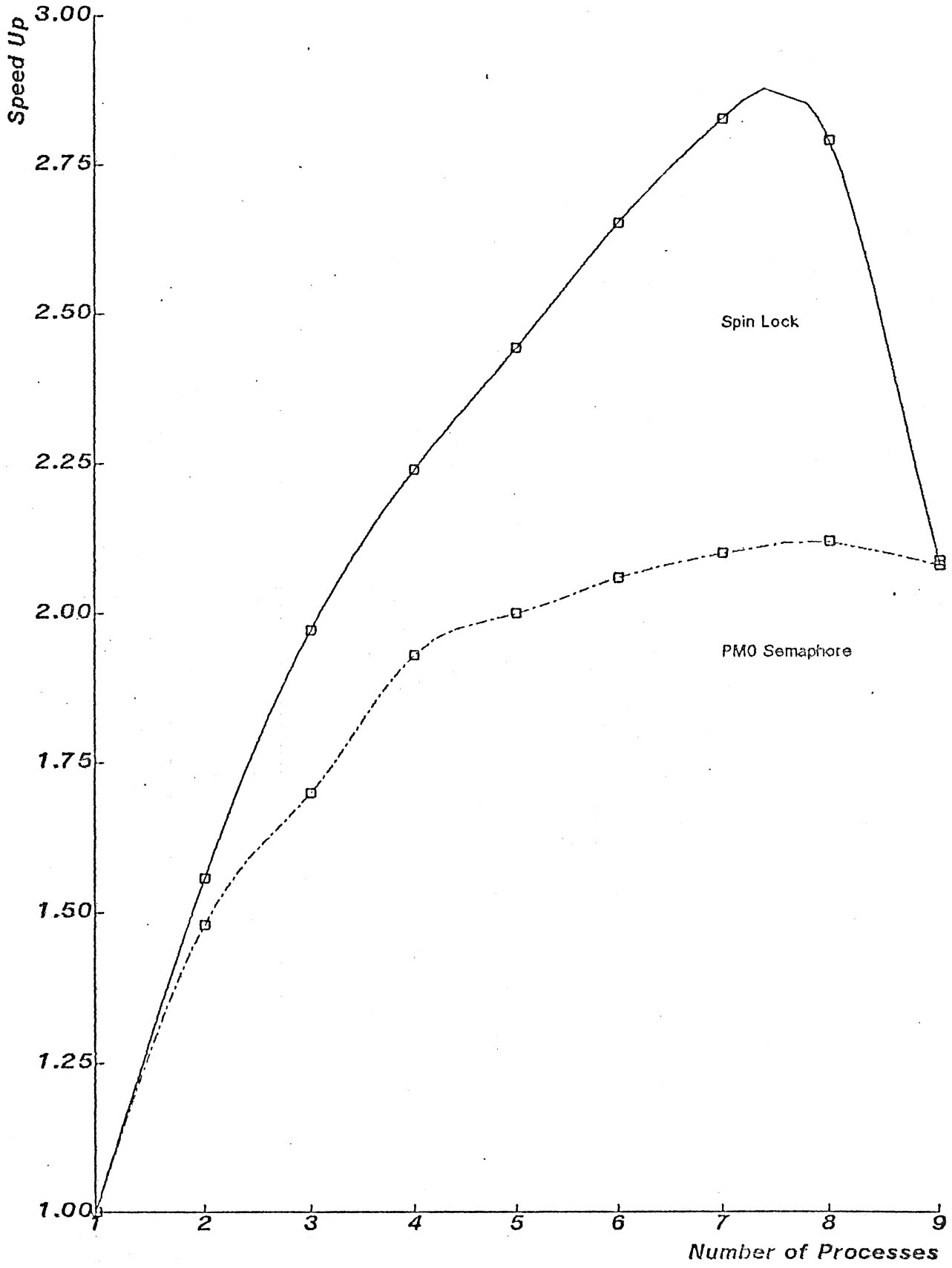


Figure 4.3 Comparison of Two Synchronization Primitives

The severe performance degradation that occurs at eight and again at nine processes for the spin-lock implementation is another instance of the NHPC problem. This time, with only seven 11/40 type processors, performance peaks at seven processes, declines slightly at eight, and then plummets from a speed up of more than 2.7 to slightly more than 2.0. Coincidently, the performance of the two implementations is nearly identical at nine processes.

However, in Figure 4.4 where the distribution is exponential there is relatively little difference in the performance of the two implementations. Because the distribution of the compute phase causes the processes to arrive at random times the PM does not become a bottleneck when the processes finish their work. When they are restarted the last one to be started is still delayed by $55 * n$ milliseconds, but since the distribution is exponential the process that must compute the function evaluation with a compute time that lies in the long tail of the distribution always finishes last. Thus the overhead of synchronization is again hidden by the MAX function that governs the STAGE time.

4.4 The Range of Usefullness for the Various Semaphores

In Figure 4.5 we have attempted to find the useful range for each of the synchronization primitives. We have graphed the performance of the rootfinder using each primitive as we vary the size of the computation phase between synchronization points. For each point, 5 cooperating processes performed 1000 total function evaluations to find 50 roots. The distribution of the function evaluation was a constant and ranged in size from 2 milliseconds to 375 milliseconds.

The NO-OVERHEAD curve is the ideal performance we would see if there were no degradation due to hardware, operating system or synchronization overheads.

The 50% line represents our threshold for adequate performance. It parallels the NO-OVERHEAD curve but represents performance exactly half of the best case. The point at which a performance curve crosses the 50% line is the threshold of usability for that synchronization primitive.

From these results we see that the spin lock is the only primitive that peforms adequately when the grain size of the compute phase is less than 15 ms. At the other extreme, all of the primitives with the exception of the initial version of the policy-module semaphore, become indistinguishable beyond 400 ms. In the region between these two endpoints the user can select the appropriate primitive to match the grain size of the computation phase. The cross-over points for the various semaphores appear in the table below.

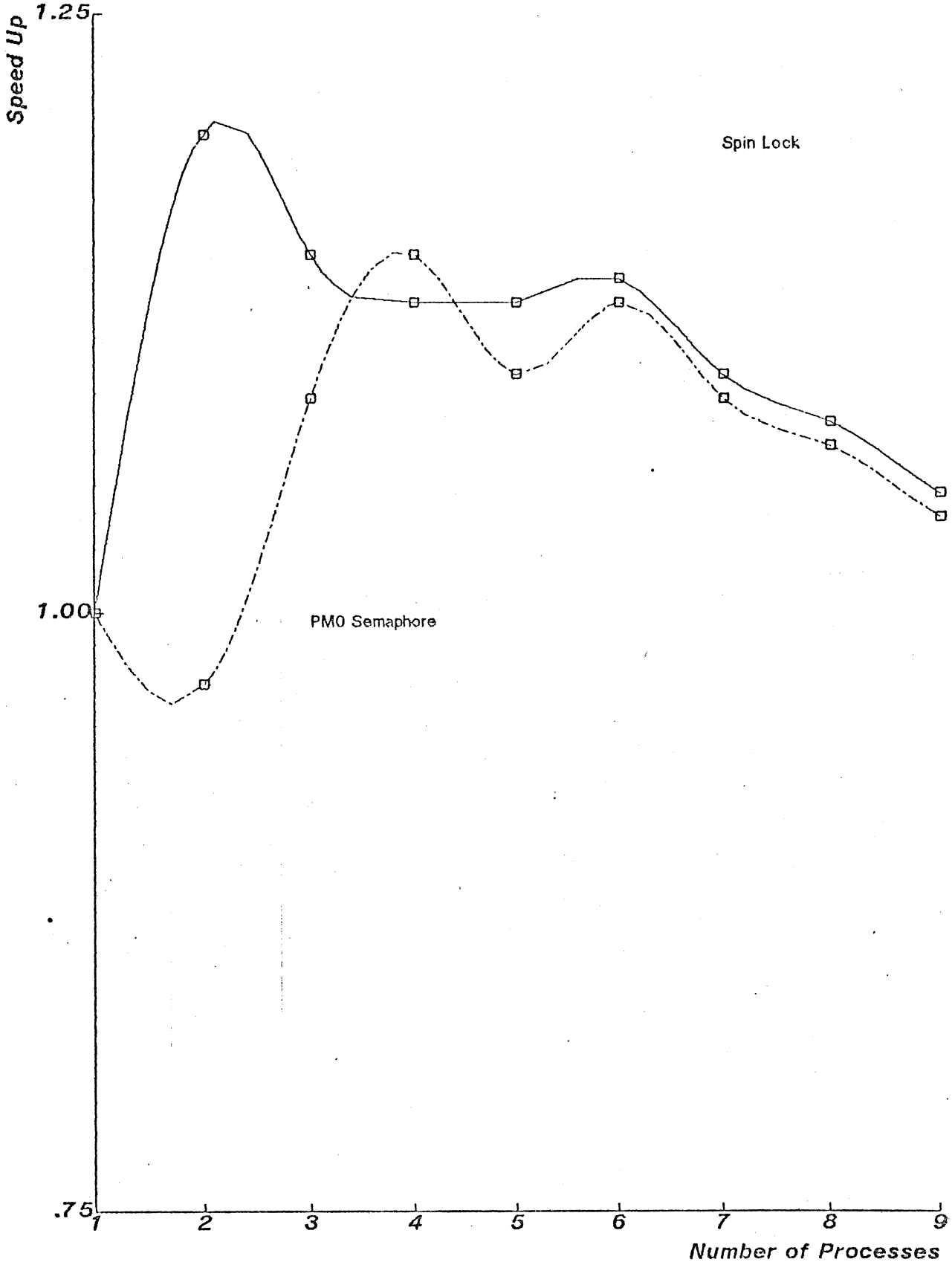


Figure 4.4 Comparison of Two Synchronization Primitives

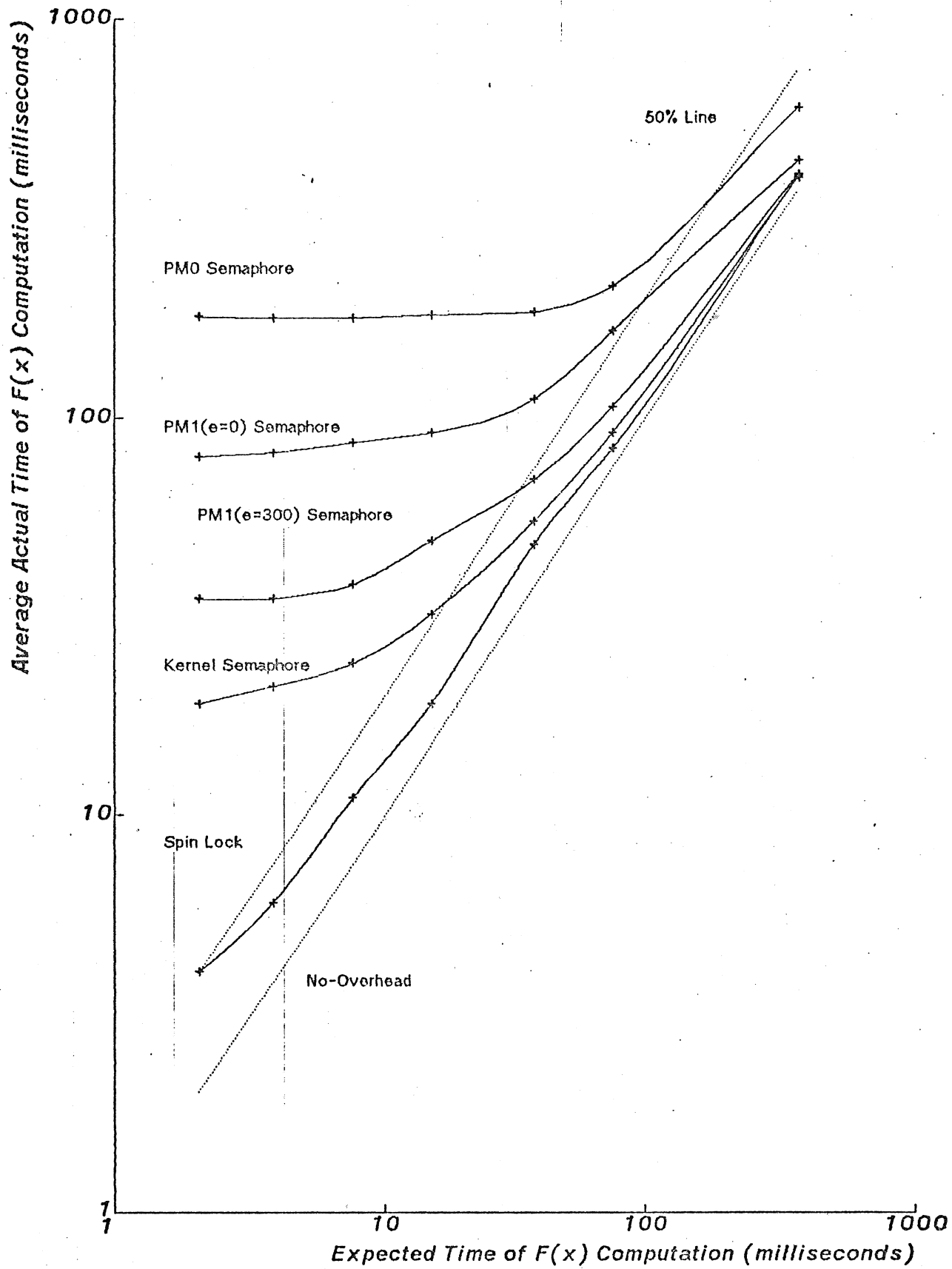


Figure 4.5 The Range of Usefullness for the Various Semaphores

Semaphore Type	Cross-over Point (msecs.)
Spin Lock	2
K-Sem	18
PM1(ϵ=300)	33
PM1(ϵ=0)	80
PM0	200

Table 4.2 Cross-over Points for the Various Semaphores

Bibliography

[Fuller 1976] Fuller S.H., Price/Performance Comparison of C.mmp and the PDP-10, 3rd Annual Symposium on Computer Architecture, Conference Proceedings, Architecture News 4,4, January 1976, pp. 195-202.

[Sauer 1977] Sauer C.H., amd Chandy K.M., The Impact of Distributions and Disciplines on Multiple Processor Systems, IBM Research Report RC 5978

[Wulf and Bell 1972] Wulf W.A., and Bell C.G., C.mmp -- A Multi-Mini-Processor, Proceedings AFIPS 1972, FJCC Vol 41. AFIPS Press, pp. 765-777.

[Wulf 1974] Wulf W.A., Cohen E., Corwin W., Jones A., Levin R., Pierson C., Pollack F., HYDRA: The Kernel of a Multiprocessor Operating System, Communications of the ACM, 17,6, 1974, pp. 337-345.

[Levin 1975] Levin R., Cohen E., Corwin W., Pollack F., Wulf W.A., Policy/Mechanism Separation in HYDRA, Proceedings of the ACM/SIGOPS Symposium on Operating Sysytems Principles, Austin Texas, November 1975, pp. 132-140.

[Stone 1973] Stone H.S. Problems of Parallel Computation, Complexity of Sequential and PArallel Numerical Algorithms ed. J.F. Traub, Academic Press 1973, pp. 1-16.

[Oleinick 1978] Oleinick P.N., The Implementation of Parallel Algorithms on a Multiprocessor, Ph.D. Thesis Carnegie-Mellon University Computer Science Dept., (Expected May 1978).

[Kung 1976] Kung H.T., Synchronized and Asynchronous Parallel Algorithms for Multiprocessors, Algorithms and Complexity: Recent Results and New Directions, ed. J.F.Traub 1976, pp. 153-200.

[Teichroew 1956] Teichroew D., Tables of Expected Values of Order Statistics and Products of Order Statistics for Samples of Size Twenty or Less from the Normal Distribution, The Annals of Mathematical Statistics 27,2, June 1956, pp 410-426.

CMU-CS-78-103

Reflections in a Pool of Processors

An Experience Report on C.mmp/Hydra

William A. Wulf
Samuel P. Harbison

Carnegie-Mellon University
Pittsburgh, PA

February, 1978

Keywords and phrases: multiprocessors, distributed computing, capabilities, operating systems, computer architecture, performance evaluation.

Abstract

This paper is a frankly subjective reflection on the successes and failures of the C.mmp project by those most intimately connected with its design, implementation, and use. It attempts to catalog and characterize the things we feel we did right and the things we did wrong. We sincerely hope that this sort of evaluation will help others who undertake similar projects.

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.

1. 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.

1.1 Multiprocessor Research at CMU

In late 1971 we at CMU decided to embark on a research program to explore multi-computer 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.

1.2 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 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.

1.3 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 kernel 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, C, L♯, and a flexible command language).

1.4 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.

1.5 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 five 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.

1.6 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.

2. 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 lists 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.

3. The Successes

3.1 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 intially 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.

3.2 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. This made the kernel more complicated.*]

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.

3.3 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.

3.4 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 occured 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 processes (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 occured. 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 "catastrophic" in this sense. We are in the midst of redesigning both hardware and software to correct these deficiencies.

3.5 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" [1] 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.

---

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

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.

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.)

4. The Failures

4.1 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 in section 3.4. 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 diagnostic 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.

4.2 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 28K 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 CDC6600-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 28K.

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.

4.3 The Partitionable System

When we first began to consider 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 it 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.

4.4 (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.

4.5 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 non-priviledged 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 lead 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 as 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 deterent 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.

5. 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.

5.1 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 department's 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 exists 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 department's 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.

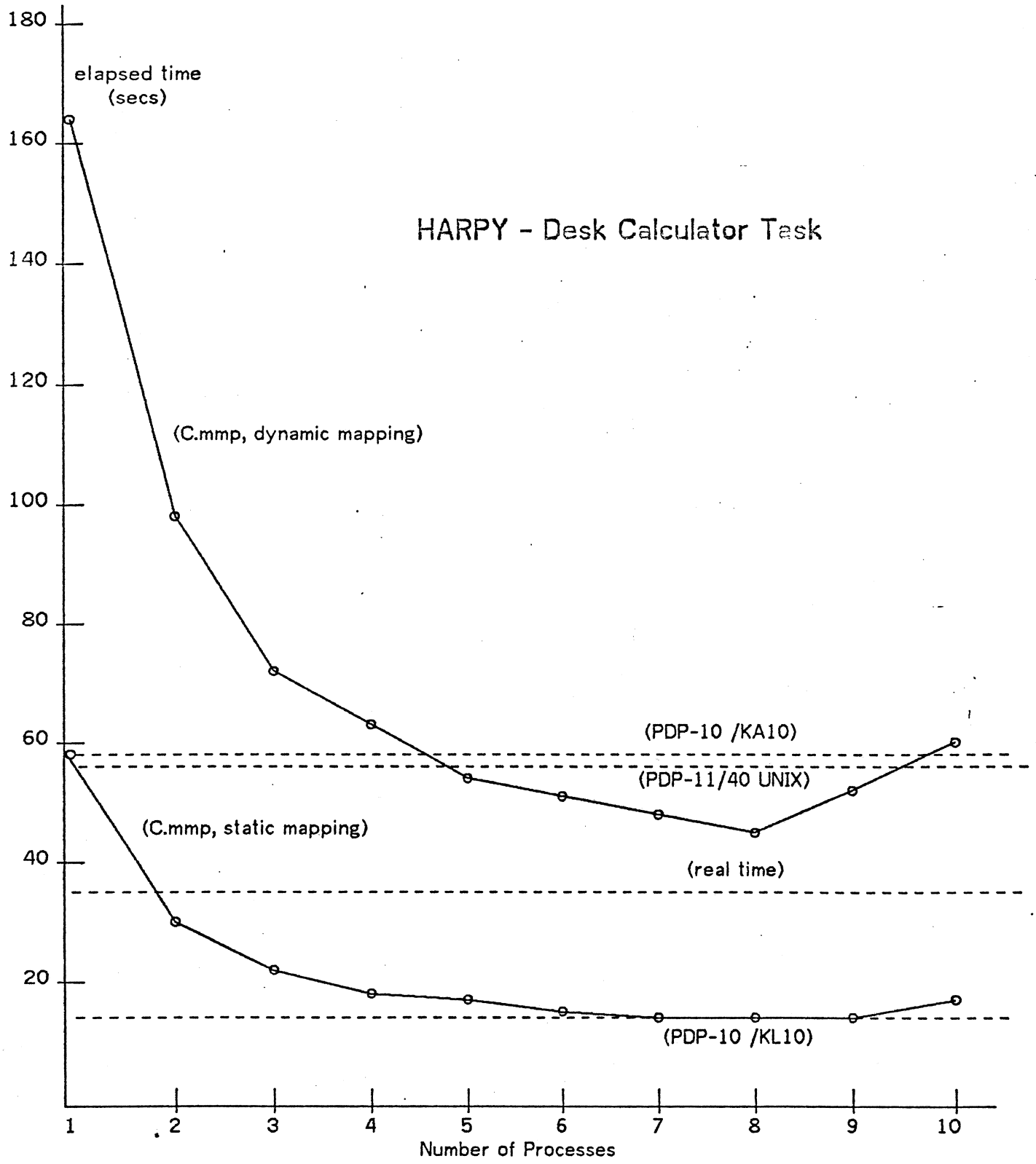


Figure 1 - A Look at the Small Address Problem

5.2 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.

Statistic	Program 1	2	3
Total time of measurement (millis)	17393	32924	20255
Nuber 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

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% of the time blocked on locks. We do not yet have any measurement of the time lost due to blocking on semaphores.

6. 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 gather 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.

6.1 Acknowledgements

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; they were:

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.

7. 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.

Almes, G. and Robertson, G., "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.)

Bell, C. G., Broadley, W., Wulf, W. A. and Newell, A., "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.

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.

Bhandarkar, D. P. and Fuller, S., "Markov Chain Models for Analyzing Memory Interference in Multiprocessors", ACM/IEEE First Annual Symposium on Computer Architecture, Dec. 1973, pp. 231-239.

Cohen, E., "Problems, Mechanisms and Solutions", Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., August 1976.

Cohen, E. and Jefferson, D., "Protection in the Hydra Operating System", *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975 pp. 141-160.

Fuller, S. and Oleinick, P., "Initial Measurements of Parallel Programs on a Multi-mini-processor", IEEE CompCon'76, September, 1976, pp. 358-363.

Fuller, S., "A Cost/Performance Comparison of C.mmp and the PDP-10", ACM/IEEE Symposium on Computer Architecture, Jan. 1976.

Fuller, S., Swan, R. and Wulf, W. A., "The Instrumentation of C.mmp: A multi-(mini)-processor", IEEE CompCon'73, 1973, pp. 177-180.

Fuller, S. H., and Stevenson, D. K., "The performance monitor for C.mmp," *11th Annual Allerton Conference*, Urbana, Illinois, October, 1973.

Fuller, S. H., "Recent developments in multiprocessor computer systems," CALCOLO Vol. XII, No. 1, June 1975, pp. 35-58.

Jones, A. K. and Wulf, W. A., "Toward the design of secure systems," *Software - Practice and Experience*, Vol 5 (1975), pp. 321-333.

Levin, R., Cohen, E., Corwin, W., Pollack, F. and Wulf, W. A., "Policy/Mechanism Separation in Hydra", *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975 pp. 132-140.

Marathe, M., and Fuller, S. H., "A study of multiprocessor contention for shared data in C.mmp," *ACM SIGMETRICS Conference*, Washington, D.C., December, 1977.

Newcomer, J., Cohen, E., Corwin, W., Jefferson, D., Lane, T., Levin, R., Pollack, F. and Wulf, W., "Hydra: Basic Kernel Reference Manual", Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., 1976.

Newell, A., Freeman, P., McCracken, D. and Robertson, G., "The Kernel Approach to Building Software Systems", Computer Science Research Review 1970-71, Computer Science Department, Carnegie-Mellon University, Pittsburgh Pa., 1971.

Newell, A., McCracken, D., and Robertson, G., "L*: An Interactive, Symbolic Implementation System," Department of Computer Science Technical Report, Carnegie-Mellon University, October, 1977.

Newell, A. and Robertson, G., "Some Issues in Programming Multi-Mini-Processors", in "Behavior Research Methods and Instrumentation", vol. 7 no. 2, March 1975, pp 75-86.

Oleinick, P. H., and Fuller, S. H., "The Implementation and Evaluation of a Parallel

Algorithm on C.mmp," Department of Computer Science Technical Report, Carnegie-Mellon University, December, 1978.

Reid, B. K. and Newcomer, J., ed., "The Hydra Songbook---A Vigilante User's Manual", Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., October 1975.

Reiner, A., and Newcomer, J., ed., "Hydra User's Manual," Department of Computer Science Technical Report, Carnegie-Mellon University, Pittsburgh, Pa., August, 1977.

Strecker, W. D., "An Analysis of the Instruction Execution Rate in Certain Computing Structures", Ph.D. Dissertation, Carnegie-Mellon University, 1971.

Wulf, W. A. and Bell, C. G., "C.mmp---A Multi-mini-processor", Proceedings of the Fall Joint Computer Conference, 1972, pp. 765-777.

Wulf, W. A. and Levin, R., "A Local Network", *Datamation*, February 1975.

Wulf, W. A., "Reliable Hardware-Software Architecture", Proceedings of the International Conference on Reliable Software, Los Angeles, 1975.

Wulf, W. A., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C. and Pollack, F., "Hydra: The Kernel of a Multiprocessor Operating System", *CACM 17*, 6 (June 1974) pp. 337-345.

Wulf, W. A., Levin, R. and Pierson, C., "Overview of the Hydra Operating System", *Proceedings of the 5th Symposium on Operating System Principles*, Austin, Texas, Nov. 1975 pp. 122-131.