CMU-CS-78-125

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

P.N. Oleinick and S.H. Fuller

Computer Science Department
Carnegie-Mellon University
June 6, 1978

Keywords: performance evaluation, multiprocessors, synchronization, parallel algorithms, cooperating processes.

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), and in part by the Office of Naval Research (Contract: N00014-77-C-0500).

ABSTRACT

C.mmp is a multi(mini) processor with up to sixteen processors. This paper presents and discusses measurements of the C.mmp system at several levels:

1. Basic hardware performance measurements
2. Runtime performance of Hydra, C.mmp's operating system
3. Overall performance of a particular application: a parallel rootfinding algorithm.

The purpose of this paper is to get a detailed look at the performance of an implementation of a parallel program on C.mmp. The rootfinding algorithm was chosen because it meets two constraints: it is a parallel algorithm with significant interprocess communication; and it is of relatively low complexity, enabling us to focus on implementation issues rather than subtleties in the algorithm itself.

Variations in processor speeds and asynchronously executing operating system functions are shown to have a detrimental effect on the rootfinder's performance. However, the most important implementation decision affecting the performance of the rootfinding program is the type of synchronization semaphore used. We define the threshold for practical application of a semaphore to be when 50% of the execution time is attributed to semaphore related overheads. Using the 50% criteria, we measured thresholds for inter-synchronization times from two milliseconds for the most primitive locks, to 200 milliseconds for the most sophisticated and flexible semaphore. During the course of these measurements, Hydra underwent several revisions and the 200 millisecond threshold was reduced to 33 milliseconds. The principal concept responsible for this performance improvement is discussed in the paper.

1. Introduction

Most papers that extol the virtues of multiprocessor computer systems cite the higher throughput and cost/performance [e.g. Sauer 1977, 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 such as memory management, process dispatching, and message passing. Upon this core an arbitrary

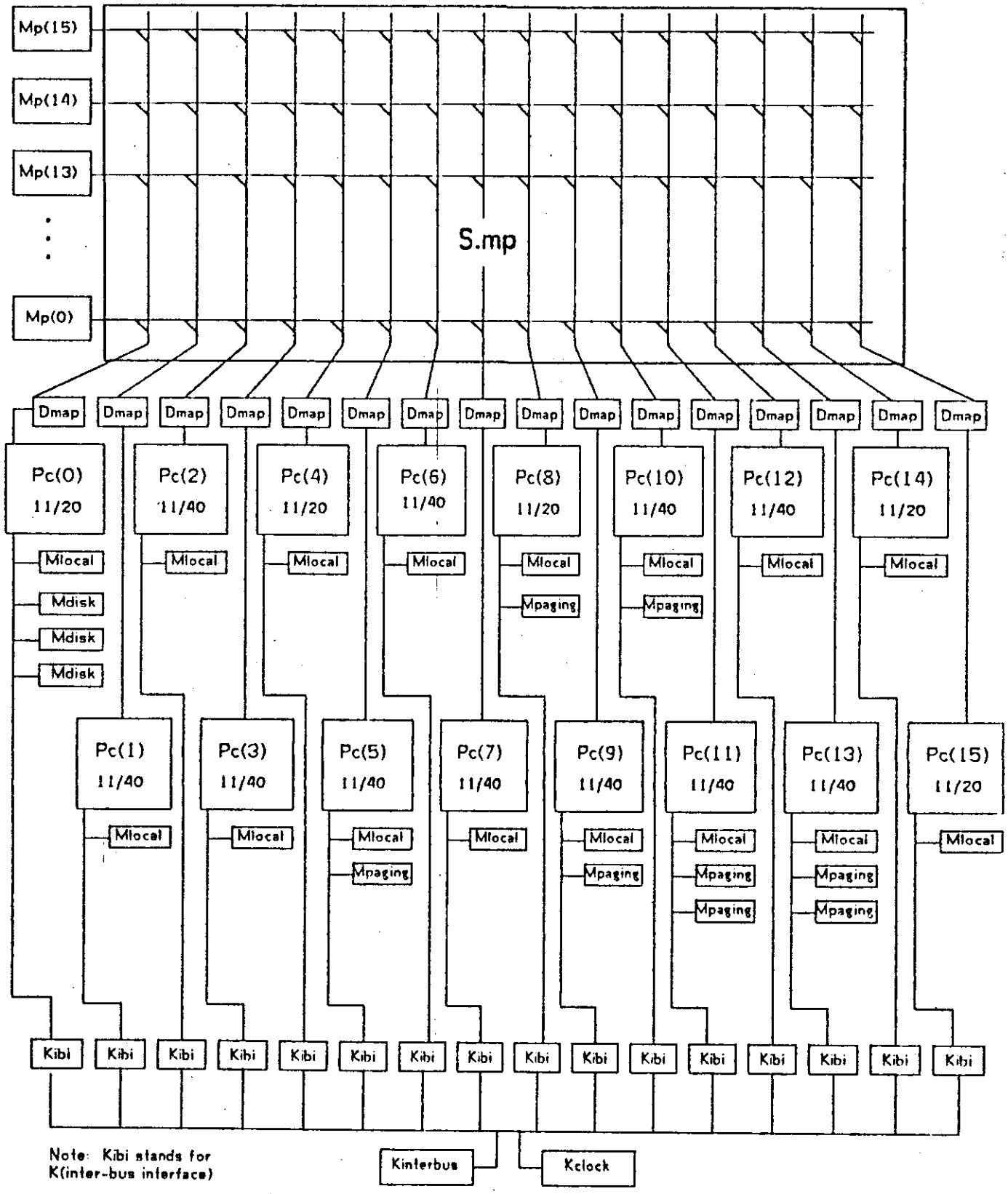

Figure 1.1 PMS Diagram of C.mmp (1977)

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 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 complex 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 less than ideal ,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

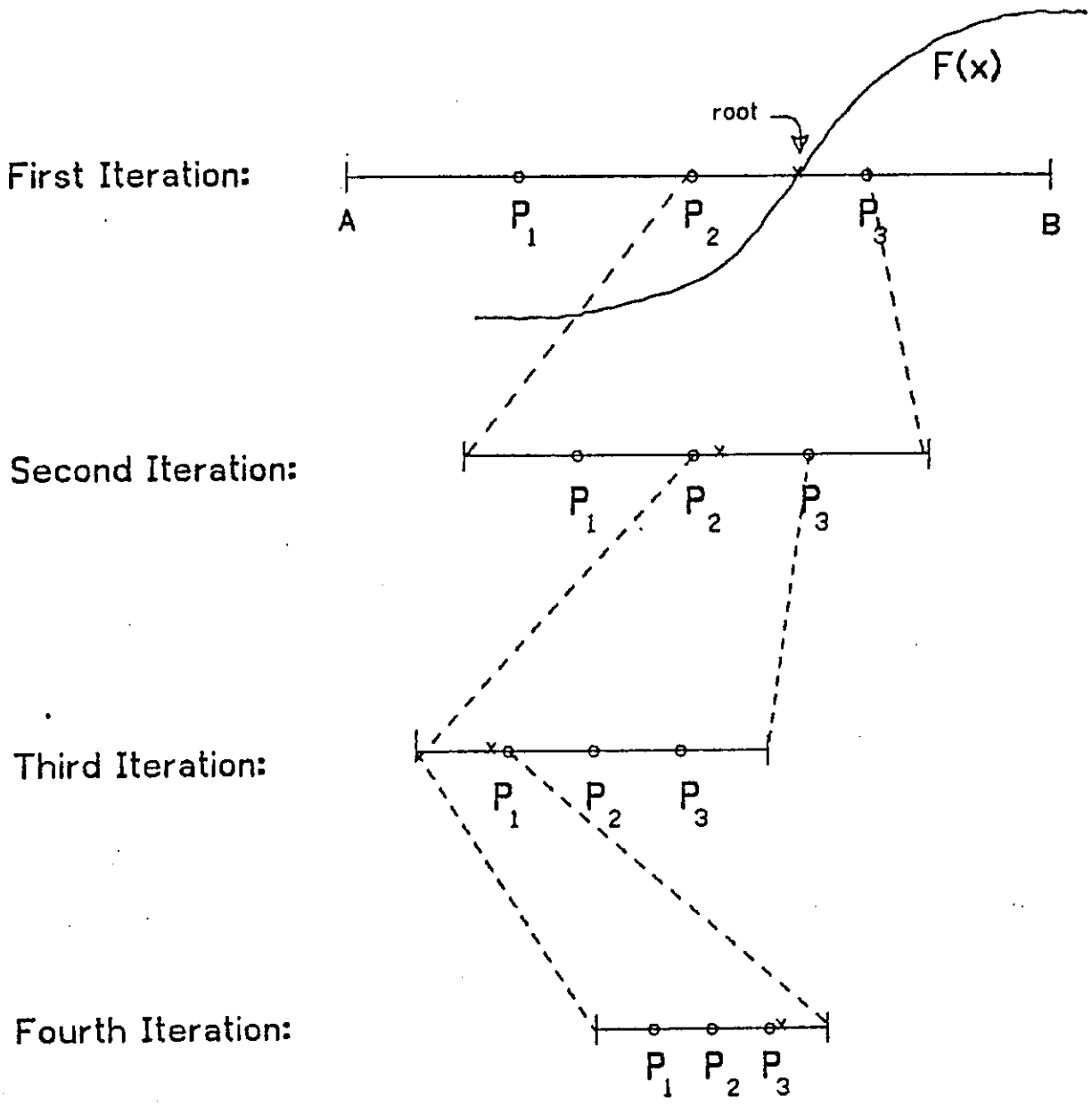

Figure 2.1 Rootfinding Program Using Three Processors

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):

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

and for larger x we used the continued fraction:

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

We selected this normal integral because it is an important transcendental function that exhibits two characteristics important to our measurement studies: it requires an extensive 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 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 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.

---

[1]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.

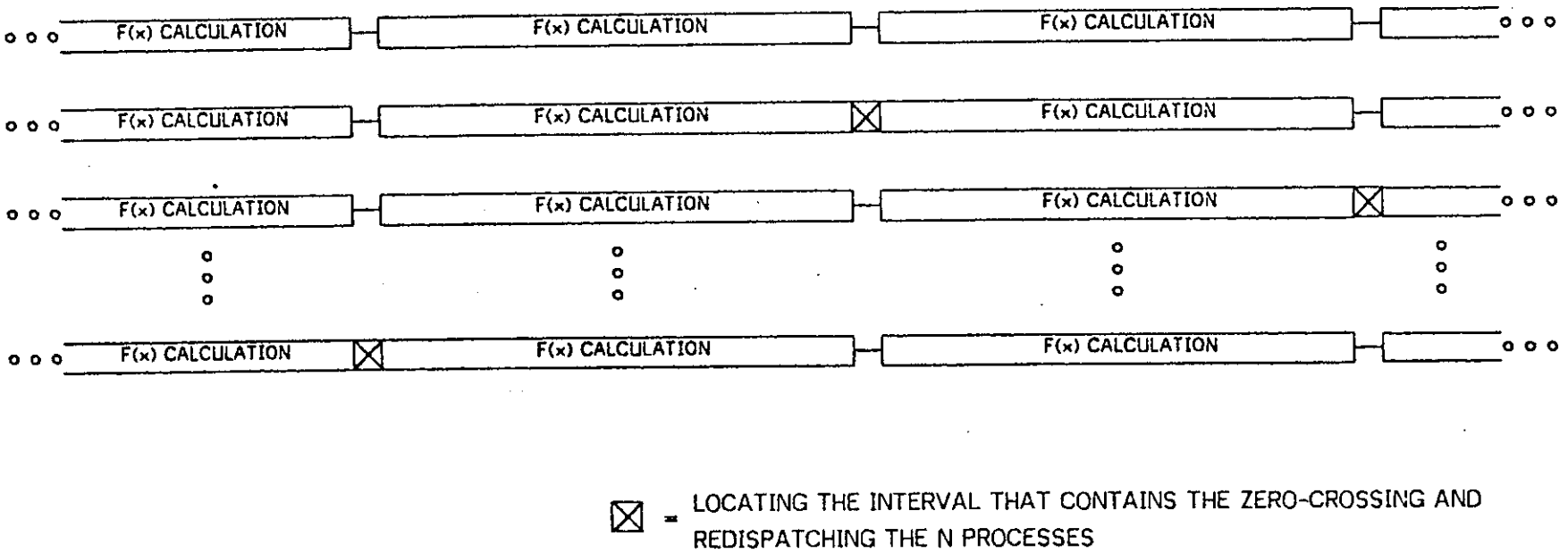

Figure 2.2 Optimal Performance of the Rootfinding Algorithm

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[1]. 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 i^{th} 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

$$T_i = MAX(t_1, t_2, t_3, \ldots , t_n) \qquad (3.1)$$

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

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

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

---

[1] On an 11/20 processor

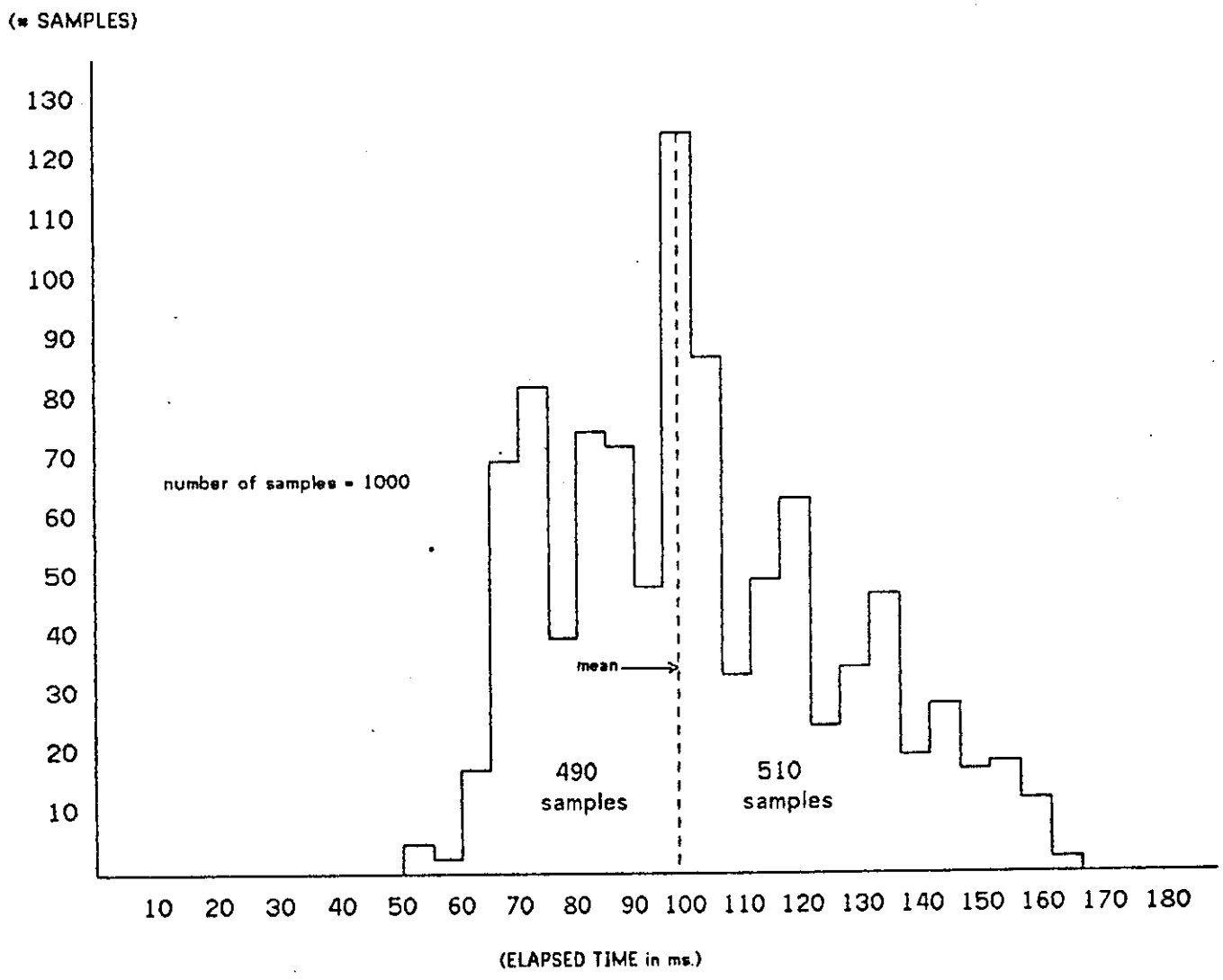

Figure 3.1 Distribution of the Time to Calculate F(x)

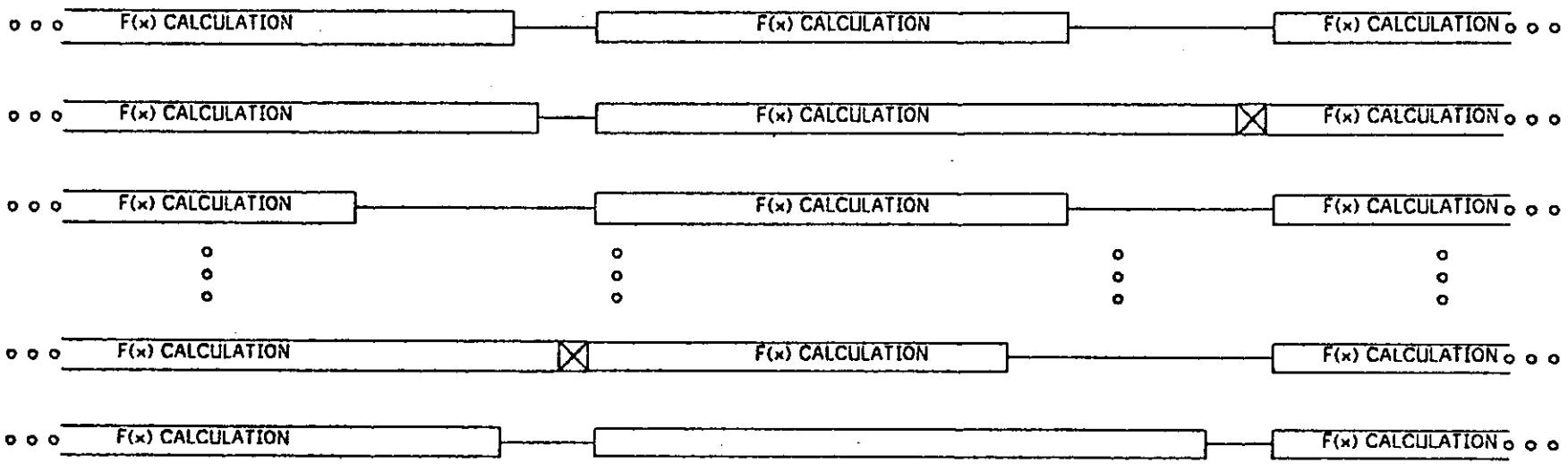

Figure 3.2 Performance Degradation Due to Variation in the F(x) Compute Time

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

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

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

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 $E(T_i) = \mu$.

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	Mean	Standard Deviation
Constant	1000	0
Normal	1000	278
Exponential	1000	1000

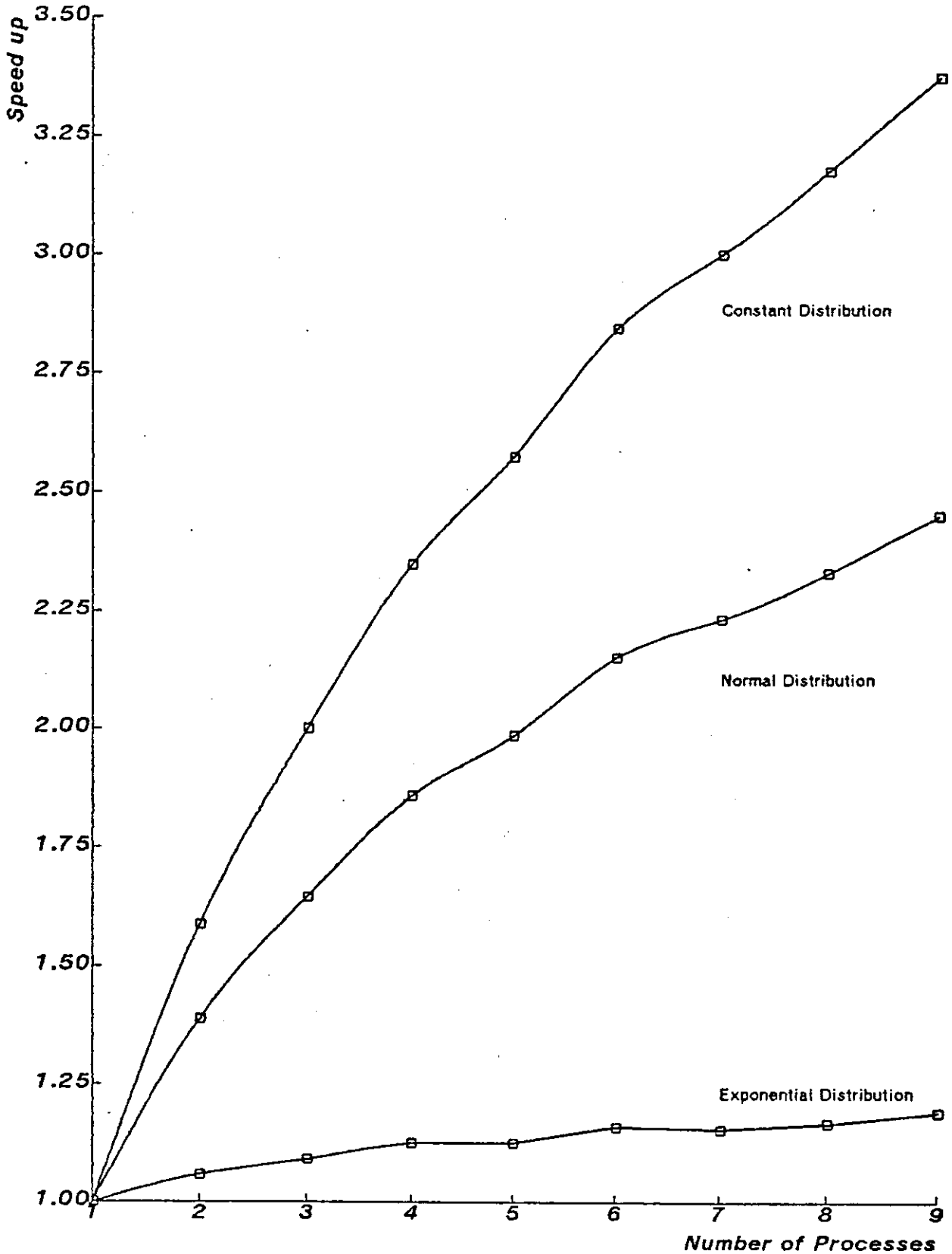

Figure 3.3 Speed up vs. Number of Processes for Ideal Multiprocessor

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 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 figure very clearly shows the impact of the variance on the performance of the rootfinding procedure. When the coefficient of variation is much greater than one, nó speedup can be 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 tọ 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. Since 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 theoretical curves.

For each single process instantiation the measured and theoretical curves are far apart. This is because any perturbation 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

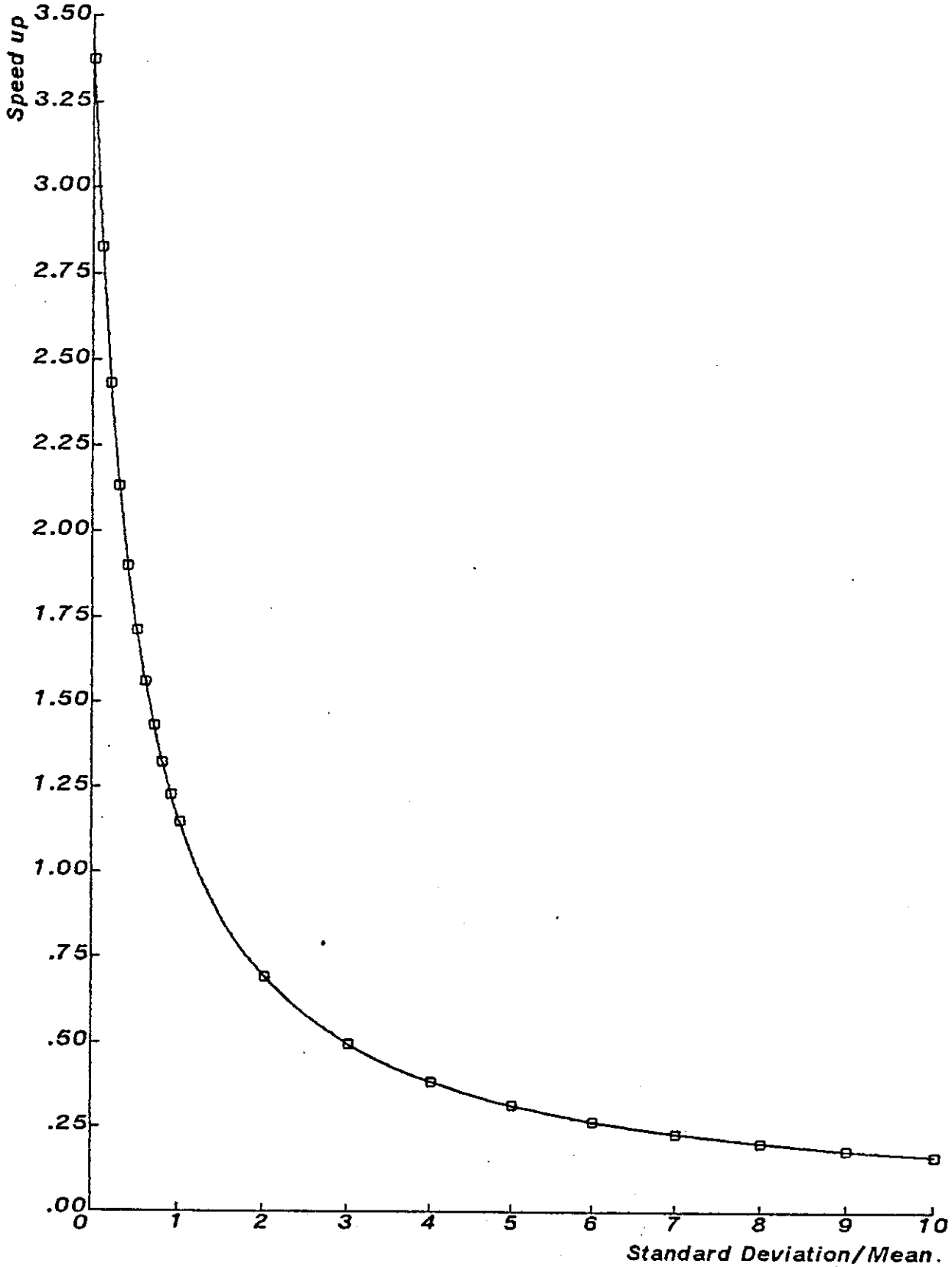

Figure 3.4 Speed Up vs. Coefficient of Variation for Nine Processes

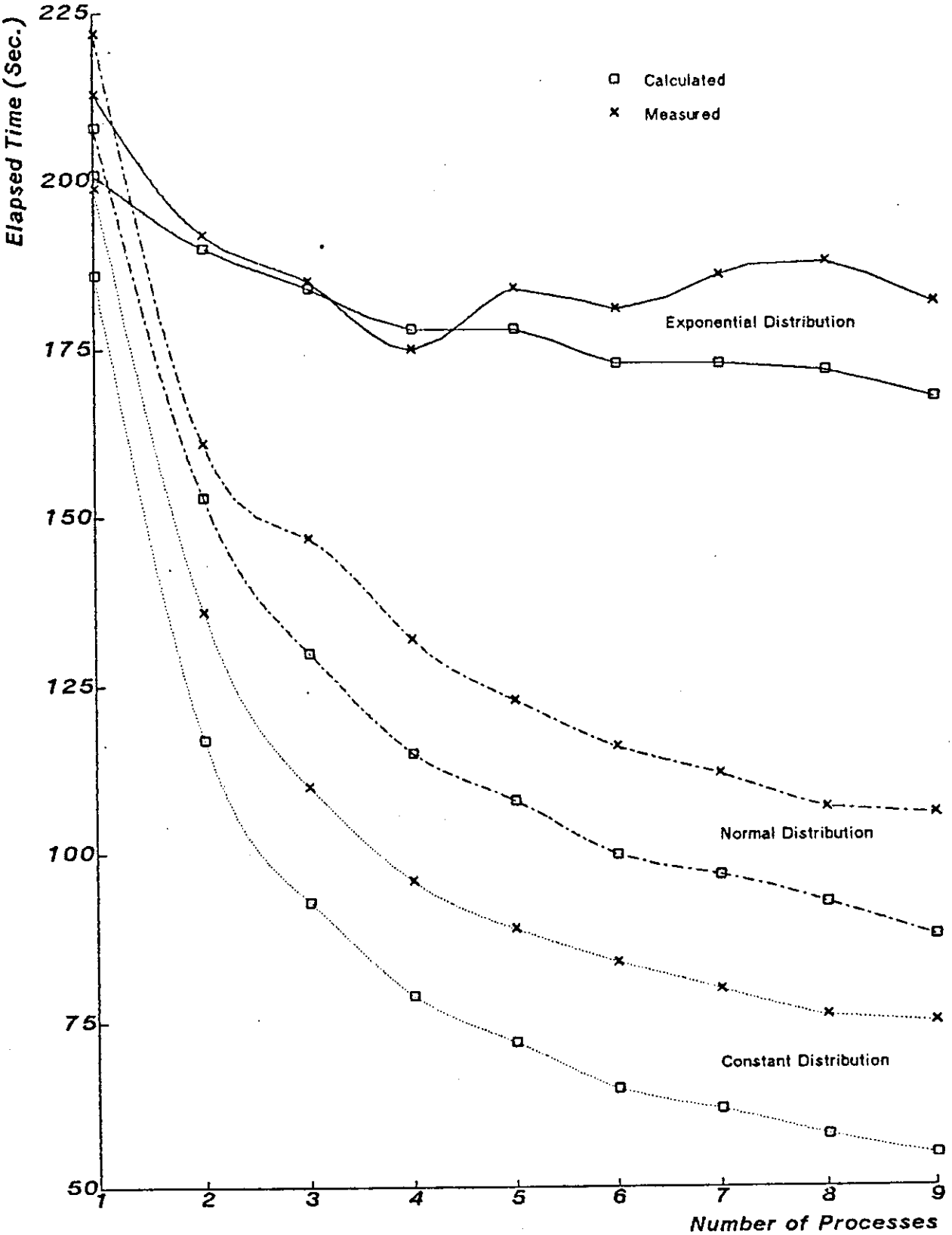

Figure 3.5 Measured Performance Compared to Calculated Performance

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. In the following sections we discuss measurements that uncover the other factors influencing performance.

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 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[1] 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].

---

[1]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 MSYNs is the number of memory requests made by the processor.

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	11/20	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 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[1]. 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.

---

[1]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.

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	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

Even among memory banks of the same technology, speed varies. 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	$1.49*10^6$	memory refs/sec.
Core	$1.71*10^6$	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[1], 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, a great deal of memory contention occurs 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, relatively little difference exists between the two tables; the one category where a noticeable change does occur is the long (> 5.0 *us.*) cycles.

[1] 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.

Less than half as many long cycles now occur 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 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 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 Histogram with Common 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 is the performance when the cooperating processes each have a copy of the code in a private memory bank.

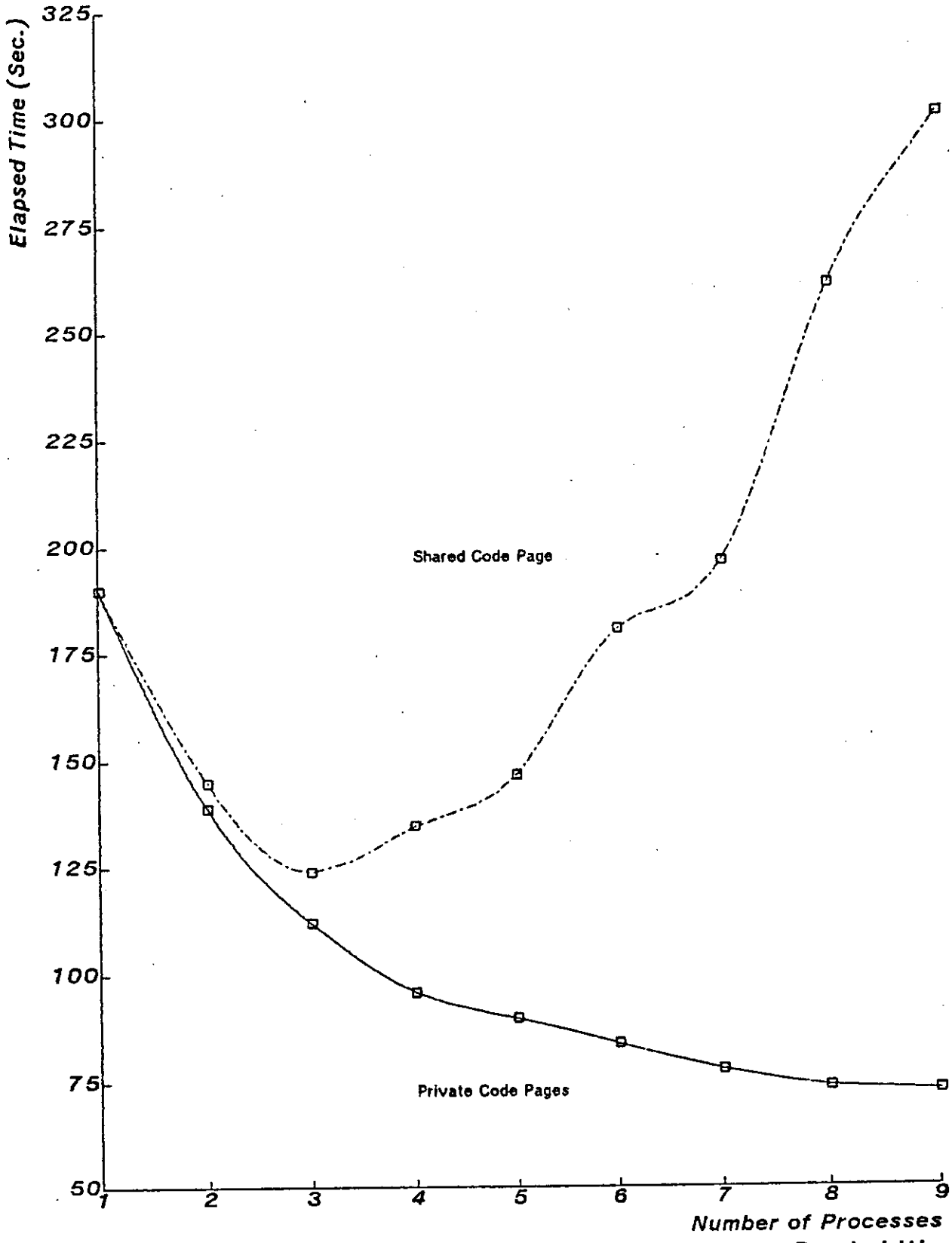

Figure 3.6 Performance Degradation Due to Finite Memory Bandwidth

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 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 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 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 Kernel 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 executing one process to executing another), semaphore activity, process starts and stops, O.S. requests (Kernel Calls) 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 n x m 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 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.

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
$\overset{n}{\lceil}$	: START OF AN INTERRUPT AT LEVEL N
⌶	: 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

Table 3.7 Tracer Symbols

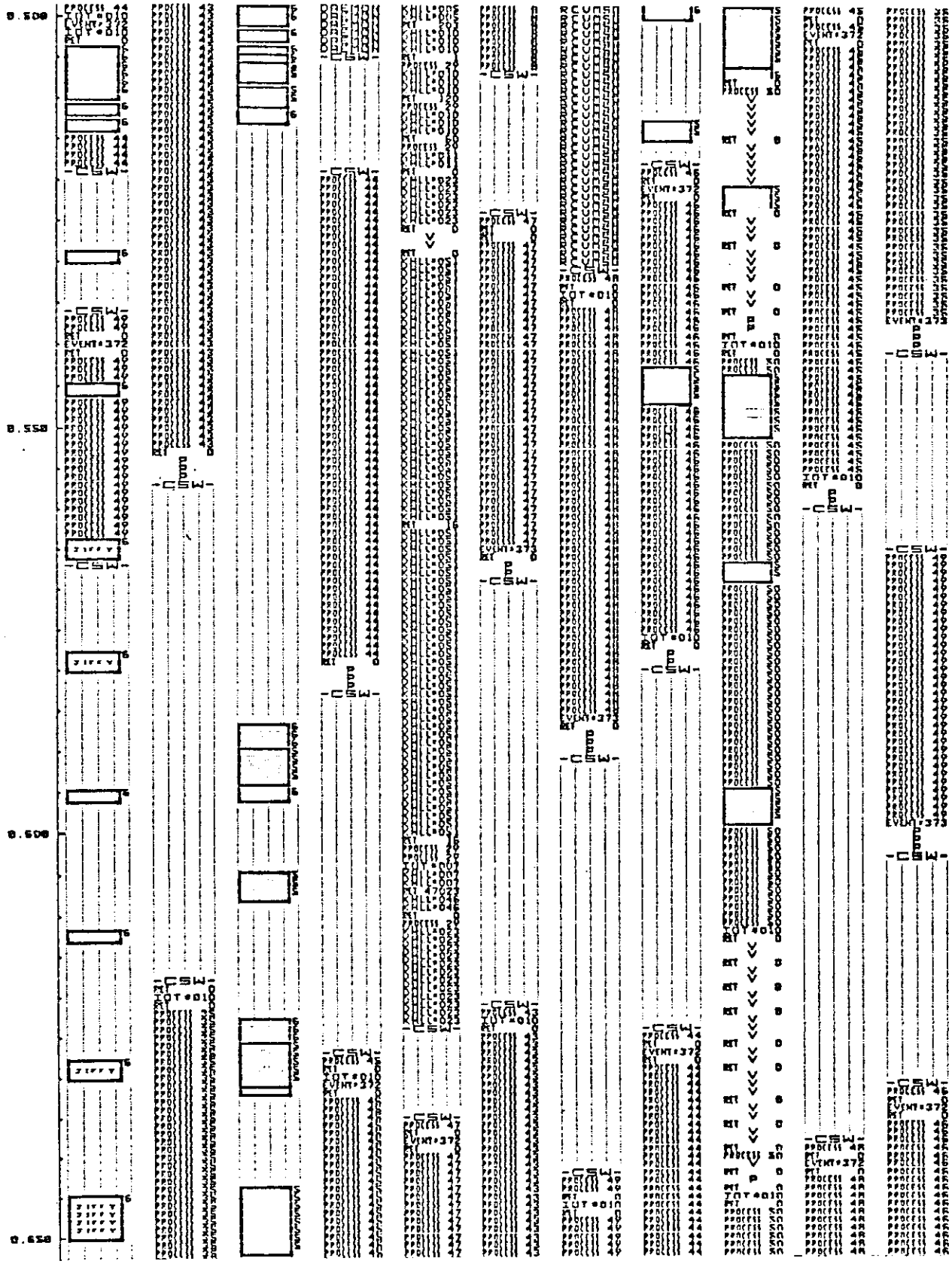

Figure 3.7a Pertubations from Interrupts

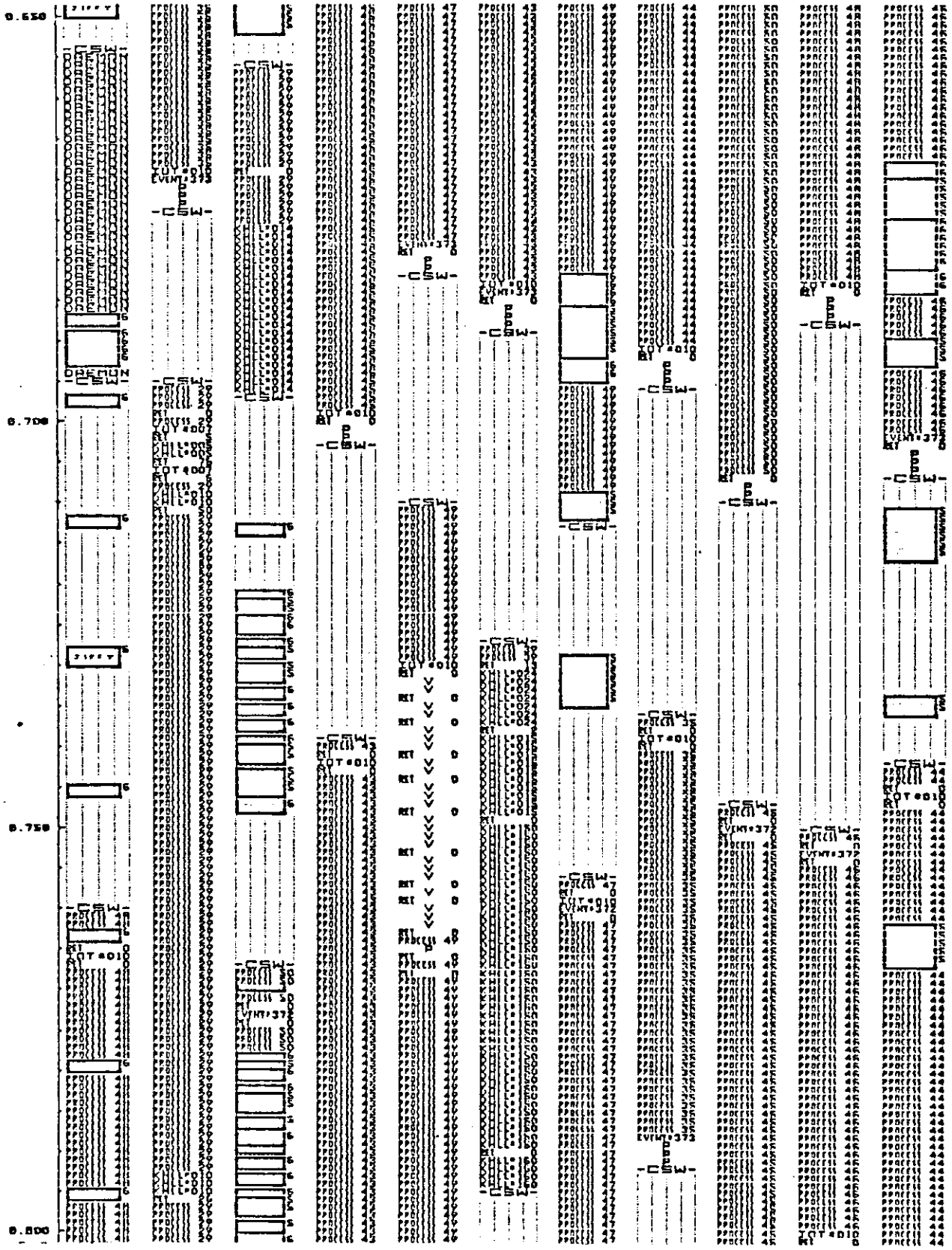

Figure 3.7b Pertubations from Interrupts

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 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 third and seventh blank columns with no execution history. Since enough processors of the prefered (11/40) type were available 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 calls 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. Moreover, by the time they get around to starting process 21, the last process that is to wake up, three 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.

Another side-effect related to the O.S. that can affect 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 request can take more than 150 milliseconds to execute. 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

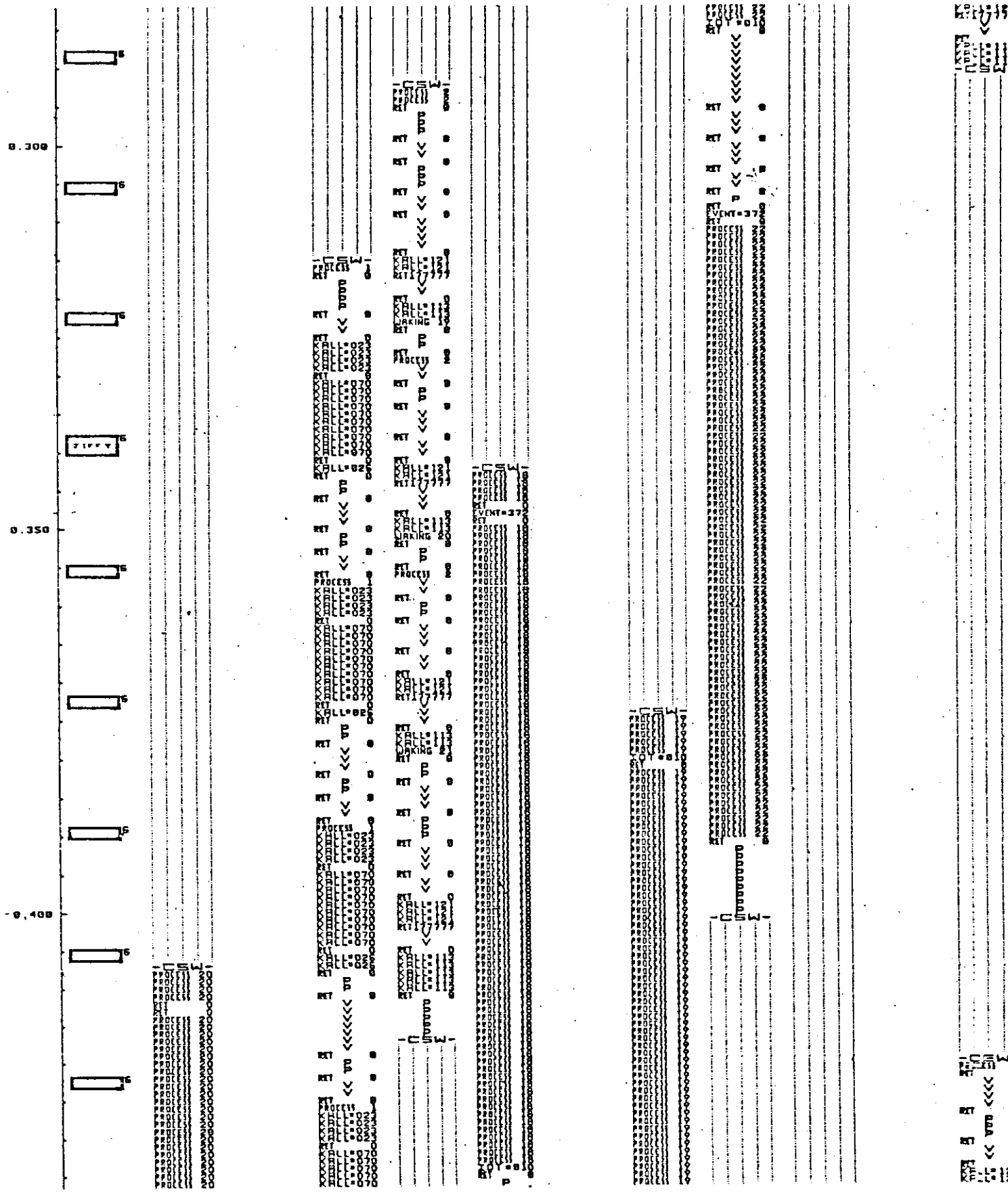

Figure 3.8a Pertubations Induced by Operating System Processes

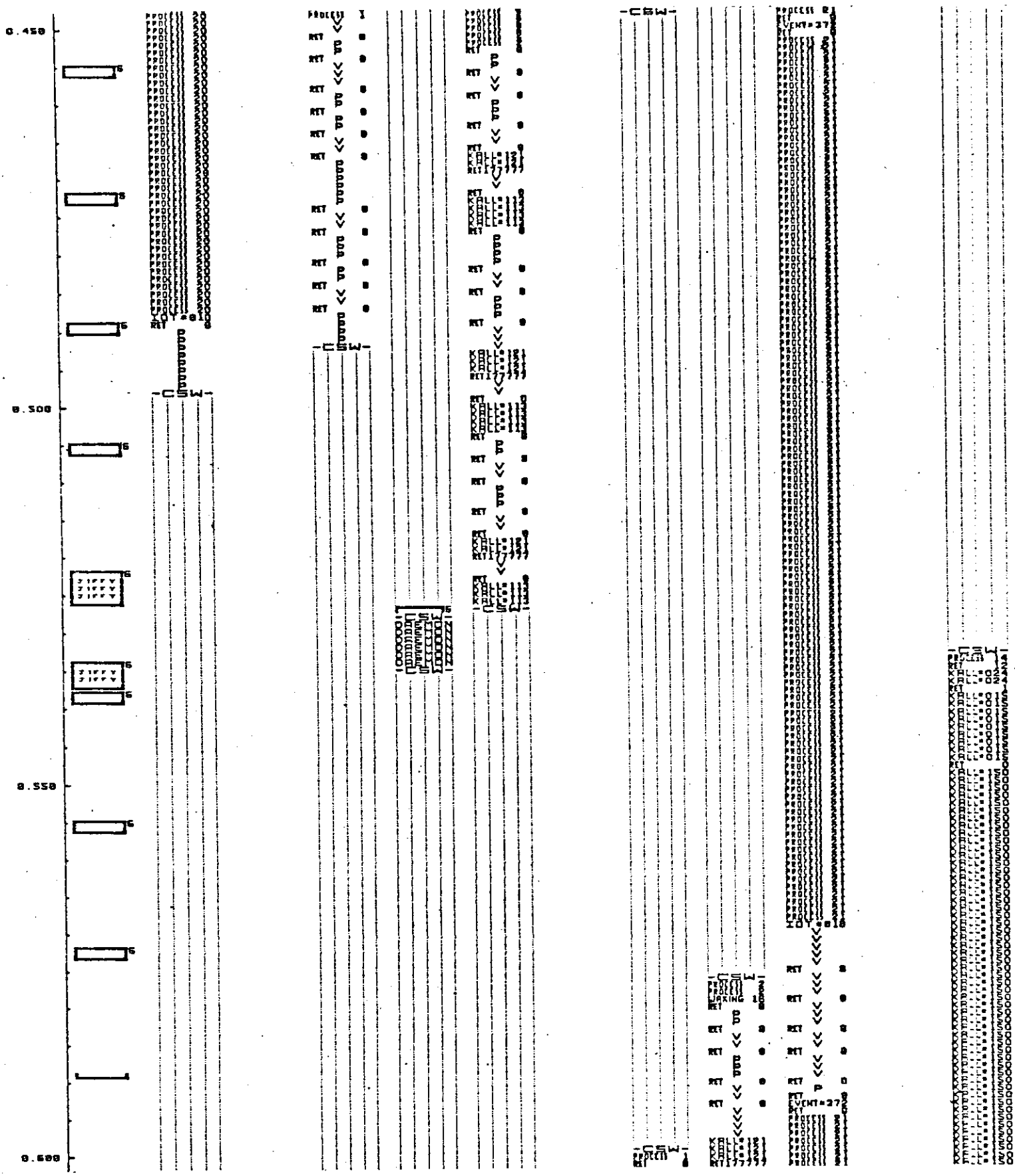

Figure 3.8b Pertubations Induced by Operating System Processes

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

The sources of performance fluctuation we have discussed can be classified into one of three types-- application, hardware, or operating system related. In the table below we rank the sources of pertubation by their potential for causing performance fluctuations. Each source is measured and the observed range calculated by dividing the maximum measurement by the minimum observed value. The larger the range, the more potential for performance fluctuation.

In the next section we eliminate several sources of pertubation in order to measure the performance of various synchronization primitives. We do this by carefully selecting processors and memory banks to execute the rootfinding program.

Rank	Type	Source	Measurement	Range
1	Application	F(x) Calculation	Function Evaluation	1 : 3.4
2	Hardware	Memory Contention	Average Cycle Length	1 : 3.0
3	Operating System	Kernel Processes	Bottlenecking of Scheduling Processes	1 : 2.8
4	Hardware	Processors	Speed	1 : 1.6
5	Operating System	I/O Devices and Interrputs	F(x) Calculation Degradation	1 : 1.3
6	Hardware	Memories	Speed	1 : 1.07

Table 3.8 The Sources of Performance Pertubation

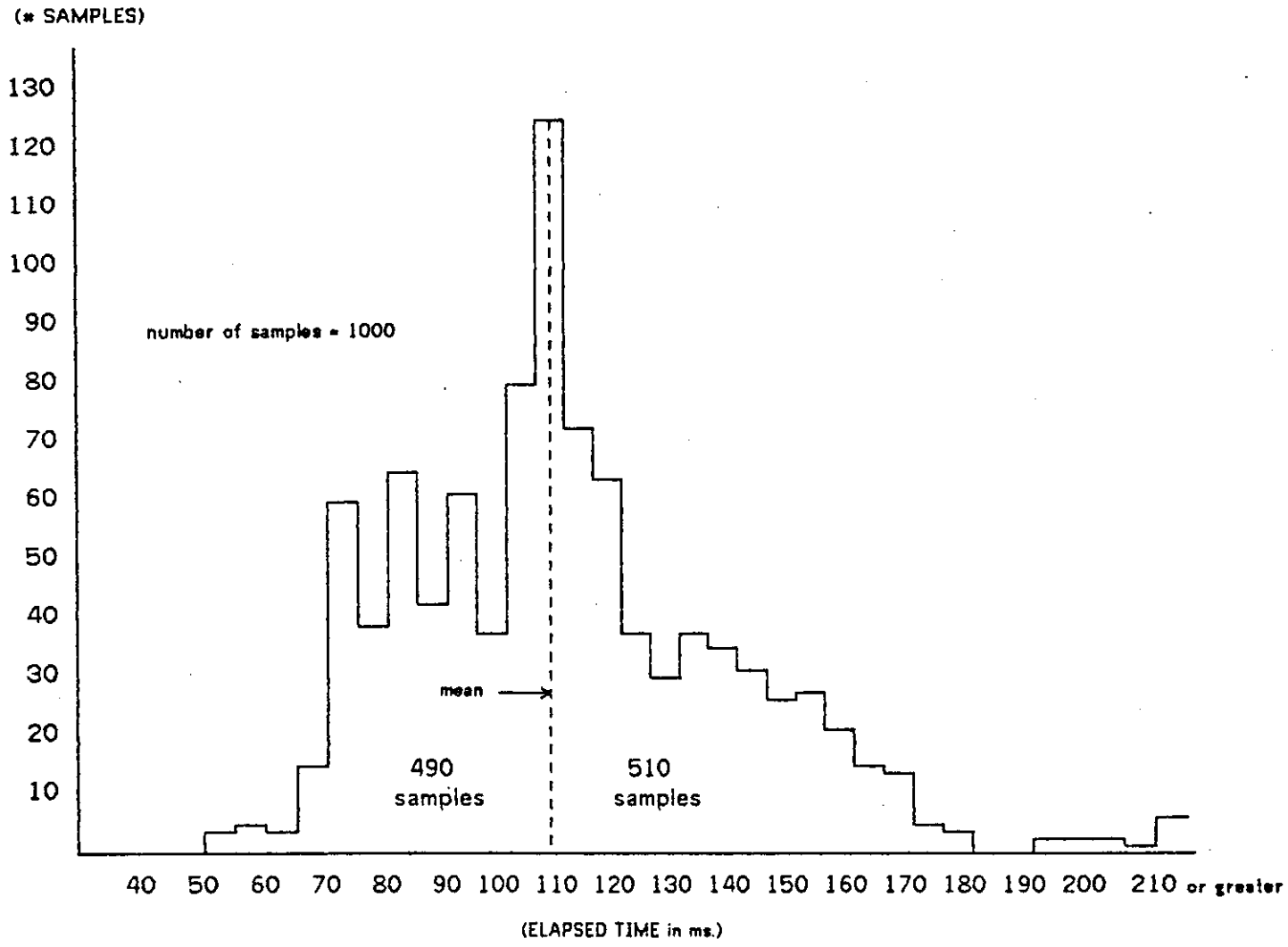

Figure 3.9 Distribution of the Time to Calculate F(x) in the Presence of HYDRA

4. The Effect of Sychronization on Performance

4.1. Introduction

Newell and Robertson[1975] identified seven programming issues for multiprocessor computer systems. Since synchronization of cooperating processes is a fundamental problem in the implementation of a parallel algorithm we will measure the performance and discuss the tradeoffs of the various synchronization mechanisms available to the C.mmp user.

Up untill 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, undesirable side effects can cause serious performance problems. We will discuss several alternative synchronization mechanisms, describe their functionality and any interesting side effects, compare 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

We 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, the overhead of synchronization has been negligable. However, 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 independently of any Hydra support and is only a tight loop in which the process continually tests a semaphore until it can set it successfully. The *P* and *V* operations are the following PDP-11 code sequences:

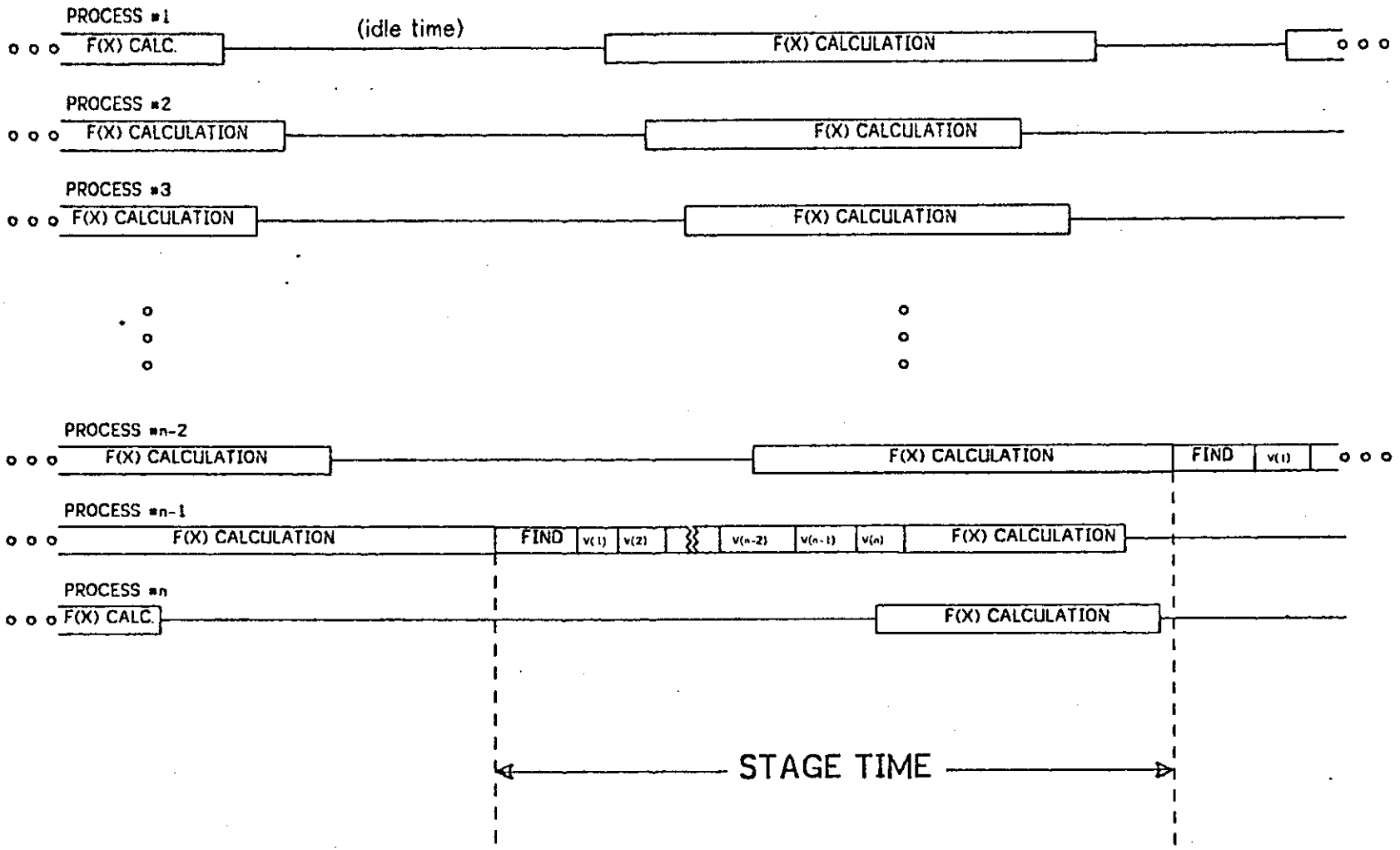

Figure 4.1 A Detailed View of the STAGE Time

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

V:      MOV #1, SEMAPHORE       ;reset SEMAPHORE = 1
```

The repeated polling of the semaphore, although extremely fast, has two very nasty characteristics.

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 consumes 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 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 by inserting a tiny delay loop in the semaphore code, i.e., decrement a register to zero before checking the semaphore. This delay will decrease the frequency of memory requests in the semaphore memory bank, but not slow the sychronization primitive appreciably. However, the primary problem still remains: a "spinning" process prevents a processor from doing 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 mechanism used by system processes. When a process blocks or wakes 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 calls. 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 remains 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 the point where it was blocked. The important attributes of the Kernel semaphore are:

- A blocked process is swapped from a processor.

- 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.

- 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, more flexibility is available in handling a blocking/waking process. The first policy that was adopted to handle blocking/waking processes was the following:

- Two PM processes 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.

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

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

- 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. 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.

Measurement	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. 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

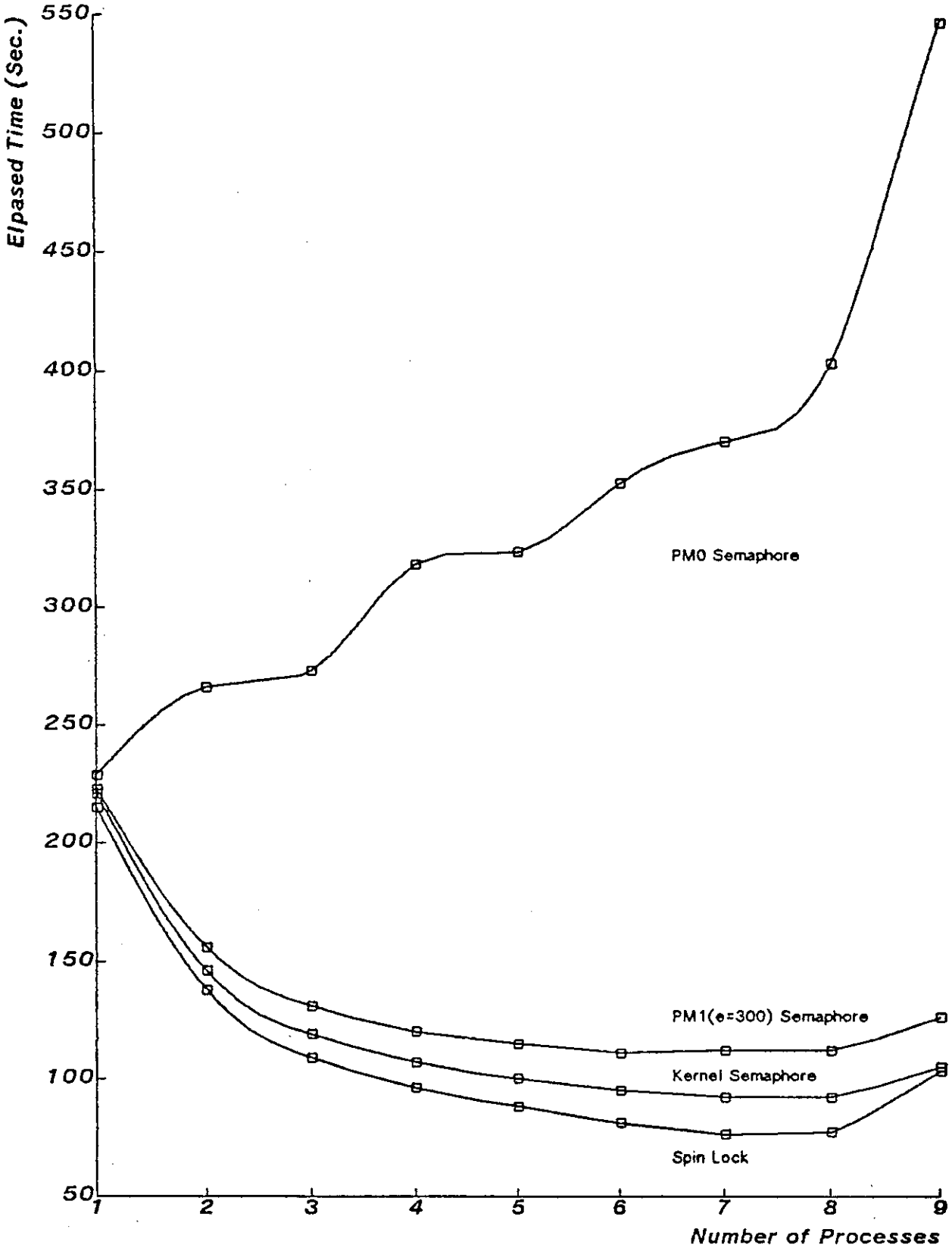

Figure 4.2 A Performance Comparison of Synchronization Primitives

plotted the elapsed time to find 50 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 is approximately Normal with mean 72 milliseconds and standard deviation 18 milliseconds[1]. 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. A process spends more time synchronizing than computing. In this instance we would be better off using a single process.

The curve for the PM1(ϵ=300) semaphore implementation depicts substantially better performance than its predecessor. Performance reaches a maximum speedup of 2.00 at six processes. No additional speedup is gained by employing more processes. Moreover, a noticeable degradation occurs 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. The STAGE time lengthed, thus 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 performance of the K-SEM implementation is about midway between the PM1(ϵ=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. The NHPC problem causes a much more severe performance degradation for this semaphore than for the others[2]. 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.

---

[1] On an 11/40 processor

[2] 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 that the two PM processes that perform the semaphore operations are almost constantly running.

4.3.3. Comparison when Compute Time is Much Greater Than 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 with the distribution a constant. The dashed curve is the performance obtained using the PM0 semaphore and the solid curve the performance 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 *serially*, 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 *serially* perform the P operations on the semaphores causing still further performance degradations.

The severe performance degradation that occurs at eight and 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. The performance of the two implementations is nearly identical at nine processes.

However, in Figure 4.4, where the distribution is exponential, relatively little difference exists between the performances 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. However, 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.

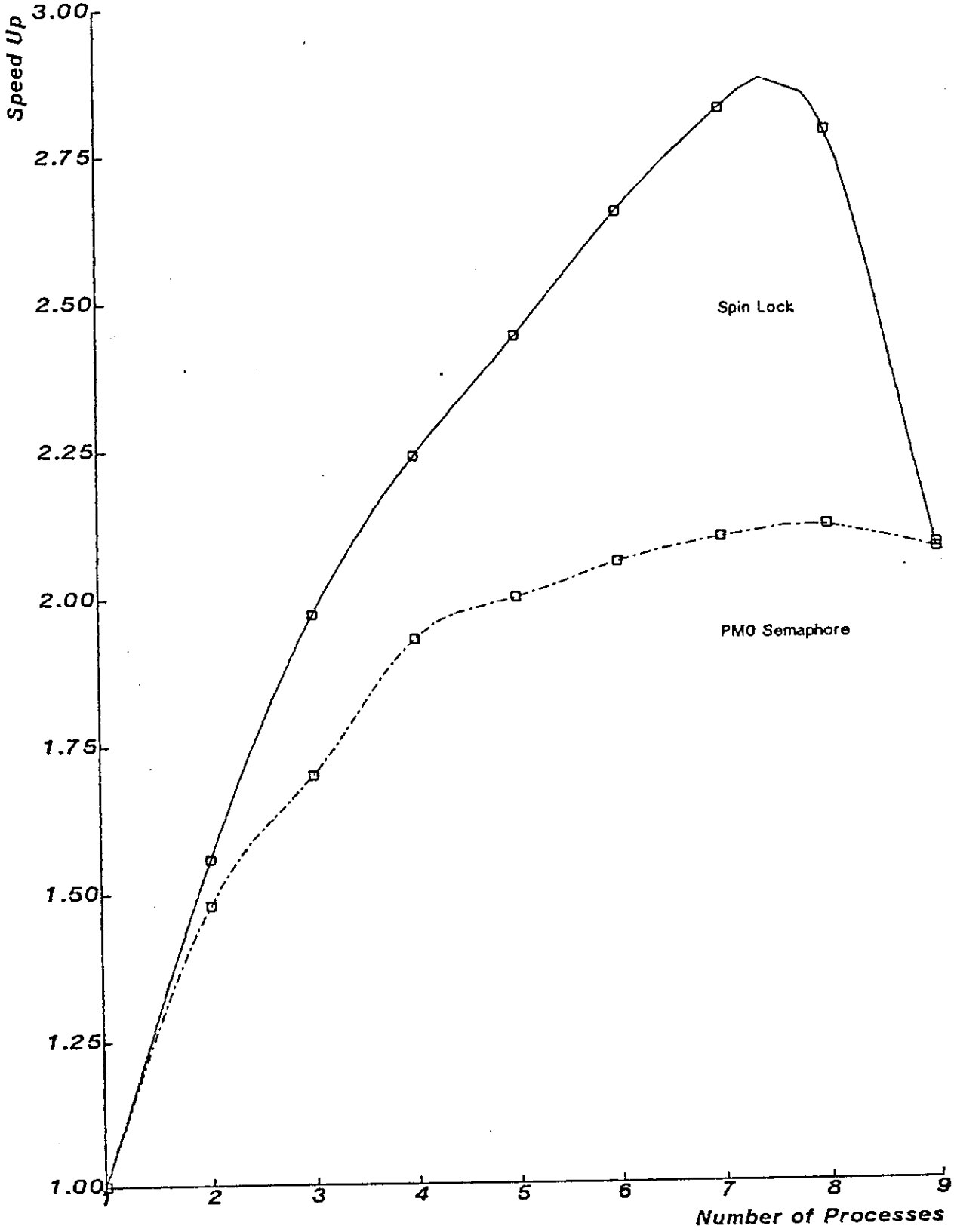

Figure 4.3 Comparison of Two Synchronization Primitives

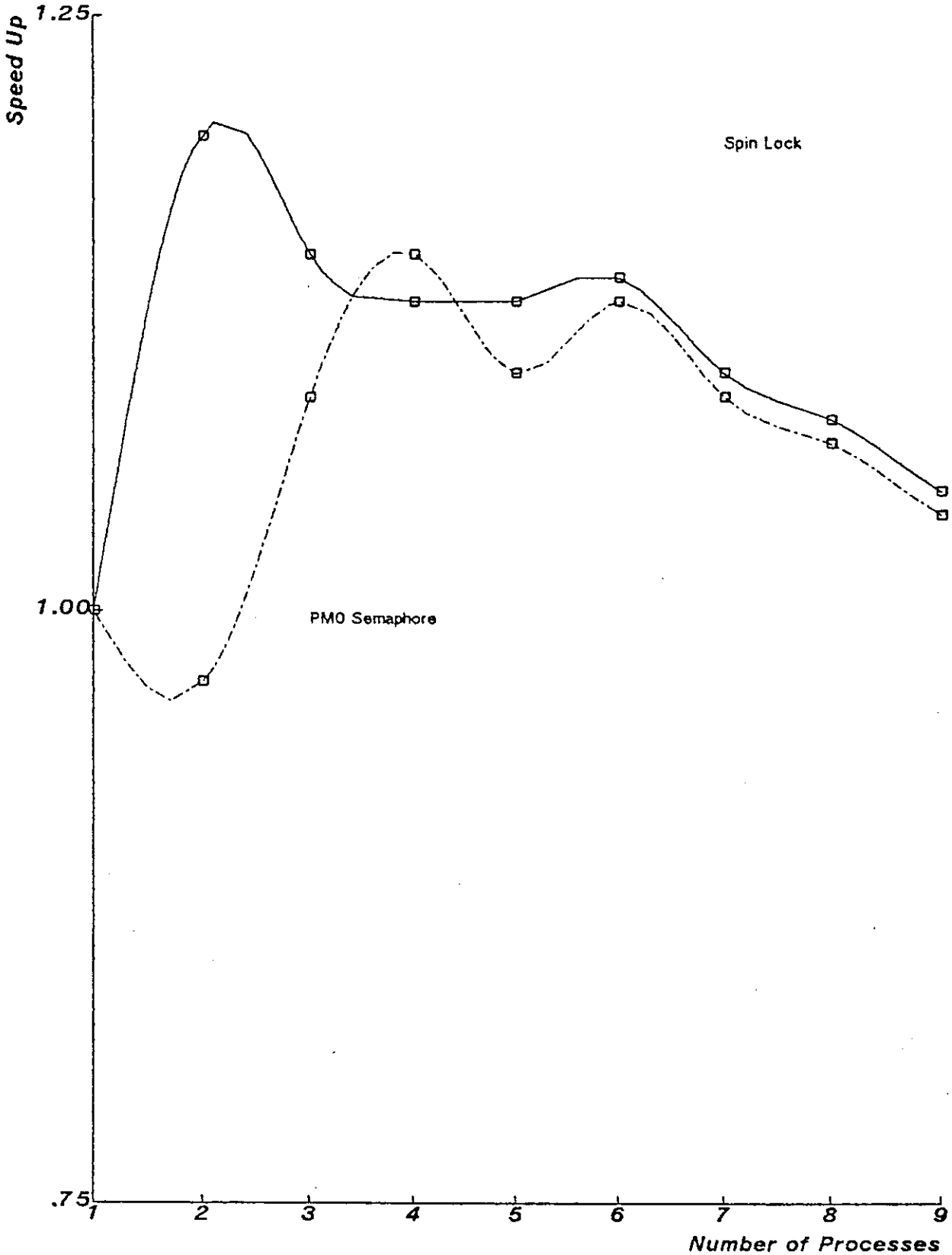

Figure 4.4 Comparison of Two Synchronization Primitives

5. Summary of Results: The Useful Range for Various Semaphores

In Figure 4.5 we have summarized the results of this investigation by graphing 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, five 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 no degradation occured 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 exactly half of the performance that would be achieved in 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 performs adequately when the length 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 length of the computation phase. The cross-over points for the various semaphores appear in the table below.

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

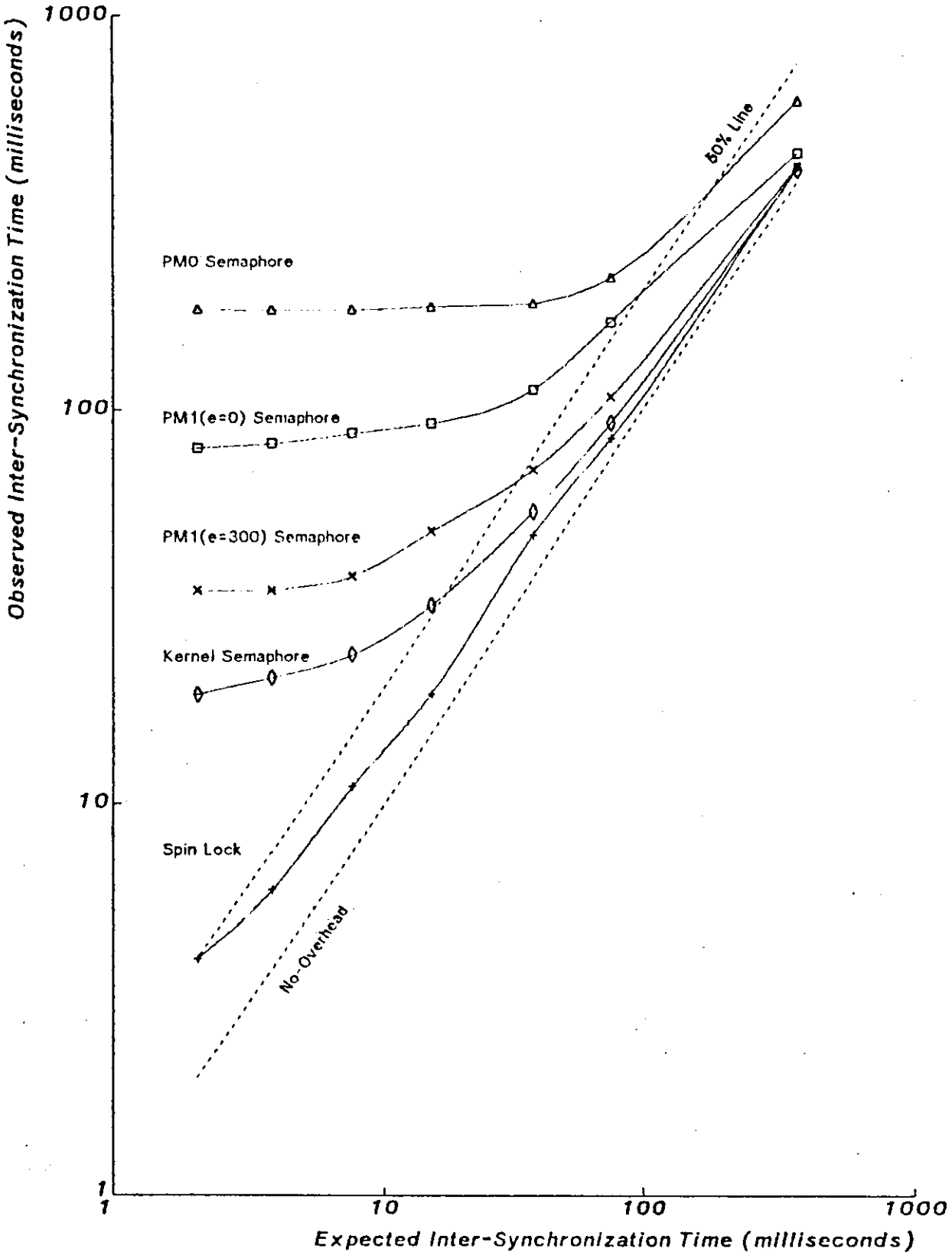

Figure 4.5 The Range of Usefulness for the Various Semaphores

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 Systems 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 July 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.

[Newell and Robertson 1975] Newell A., and Robertson G., Some Issues in Programmin Multi-Mini-Processors, Tech. Rep., Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa., January 1975

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-78-125	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE IMPLEMENTATION AND EVALUATION OF A PARALLEL ALGORITHM ON C.MMP		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) P.N. Oleinick and S.H. Fuller		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0500
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE June 6, 1978 13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) same as above		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

C.mmp is a multi(mini) processor with up to sixteen processors. This paper presents and discusses measurements of the C.mmp system at several levels:

1. Basic hardware performance measurements

2. Runtime performance of Hydra, C.mmp's operating system

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

3. Overall performance of a particular application: a parallel rootfinding algorithm.

The purpose ot this paper is to get a detailed look at the performance of an implementation of a parallel program on C.mmp. The rootfinding algorithm was chosen because it meets two constraints: it is a parallel algorithm with significant interprocess communication; and it is of relatively low complexity, enabling us to focus on implementation issues rather than subtleties in the algorithm itself.

Variations in processor speeds and asynchronously executing operating system functions are shown to have a detrimental effect on the rootfinder's performance. However, the most important implementation decision affecting the performance of the rootfinding program is the type of synchronization semaphore used. We define the threshold for practical application of a semaphore to be when 50% of the execution time is attributed to semaphore related overheads. Using the 50% criteria, we measured thresholds for inter-synchronization times from two milliseconds for the most primitive locks, to 200 milliseconds for the most sophisticated and flexible semaphore. During the course of these measurements, Hydra underwent several revisions and the 200 millisecond threshold was reduced to 33 milliseconds. The principal concept responsible for this performance improvement is discussed in the paper.